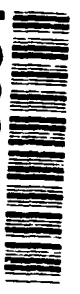
The RHET System: A Sequence of Self-Guided Tutorials

James F. Allen and Bradford W. Miller

Technical Report 325
July 1991

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

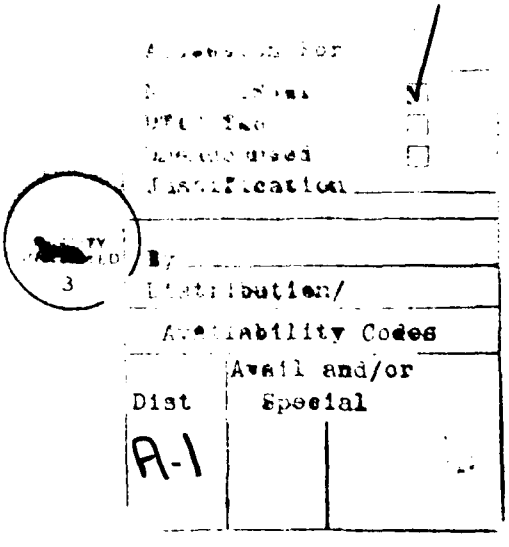


The RHET System:
A Sequence of Self-Guided Tutorials

James F. Allen and Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 325
July 1991



Abstract

The RHET system is a knowledge representation tool that is intended to support the development of advanced prototype natural language understanding and planning systems. It is what is currently called a "hybrid" representation, which consists of a set of separately defined specialized reasoning systems that are presented to the user within a single uniform framework. It can be used as a horn-clause based theorem proving system, or it can be used as a rich frame-based representation, or used in any way falling between these styles of use. The primary specialized reasoning components include a type hierarchy system, an equality reasoning system, a temporal reasoning system, and a hierarchical context mechanism that support reasoning about different agent's beliefs as well as hypothetical reasoning. This report provides a sequence of tutorials each demonstrating a major feature of the system.

The RHET System: A Sequence of Guided Tutorials

The RHET system is a knowledge representation tool that is intended to support the development of advanced prototype natural language understanding and planning systems. It is what is currently called a "hybrid" representation, which consists of a set of separately defined specialized reasoning systems that are presented to the user within a single uniform framework. It can be used as a horn-clause based theorem proving system, or it can be used as a rich frame-based representation, or used in any way falling between these styles of use. The primary specialized reasoning components include a type hierarchy system, an equality reasoning system, a temporal reasoning system, and a hierarchical context mechanism that support reasoning about different agent's beliefs as well as hypothetical reasoning.

RHET is built in Common Lisp [Steele, 1990] and has many features and options. This report provides a sequence of tutorials each demonstrating a major feature of the system. The first section introduces RHET.1, a subset of RHET that can be viewed as a variant of PROLOG, and is the basis for all subsequent sections. Once you are familiar with the RHET.1 and RHET.2 subsets, most of the other tutorials can be examined independently of the others. This way, you can become familiar with the features of RHET that you are interested in without having to read through many other features that are not of interest. Each tutorial presents the new concepts, defines the syntax for the new parts of the RHET, describes the basic functions in the user interface, presents examples that illustrate the new techniques and capabilities, and finally describes any other functions and built-in predicates that are useful in certain applications.

Appendix A describes how to install and run RHET in the Symbolics environment and in Allegro Common Lisp. Further details on RHET can be found in the reference manual [Miller 1990a] and the programmer's guide [Miller 1990b]. This report documents version 19, while the reference manual and programmers guide document slightly older versions. In cases of conflict, this report should be believed. Appendix B indicates some of the changes with version 19.

1. RHET.1: RHET as PROLOG

1.1 Introduction

The core of RHET is a representation of Horn clauses that provides a uniform interface to all of the RHET subsystems. In this section we introduce the part of the RHET language and the user interface that corresponds to a fairly traditional PROLOG system. While it is possible to simply use RHET as a version of PROLOG, there is significant computational overhead from the rest of the system. So if you don't need some of the features provided by RHET, you should use a commercial PROLOG. This comparison with PROLOG, however, allows us to introduce RHET quickly and provides a basis for all the following tutorials. If you are not familiar with PROLOG, we recommend one of the standard texts on the language, such as Clocksin and Mellish [1981] or Sterling and Shapiro [1986].

Knowledge in RHET is expressed as Horn Clauses, which are expressions of the form [$form_0$ *index* $form_1$ ... $form_N$]. $Form_0$ will be referred to as the *head* of the clause. Forms are in prefix form and consist of a predicate name followed by a list of arguments, i.e., [*pred-name* arg_0

... arg_n] and correspond to literals in the theorem proving literature. A RHET index is any atom beginning with the character "<". Indices are used only to help organize the database for retrieval and update and have no effect on any RHET inference process.

Variables in RHET are of the form *?name*typename*, as in *?x*T-U*. For now, we only need to distinguish between variables ranging over RHET terms (of type *T-U) and variables ranging over lisp objects (of type *T-LISP). Only the first occurrence of a variable in a horn clause needs to indicate its type. The following are RHET *axioms*:

```
[[Parent ?x*T-U ?y*T-U] <INDEX1 [Father ?x ?y]]
        ; ?x is a parent of ?y if ?x is the father of ?y, with index "<INDEX1"
[[Parent ?x*T-U ?y*T-U] <INDEX2 [Mother ?x ?y]]
        ; ?x is a parent of ?y if ?x is the mother of ?y; with index "<INDEX2"
```

RHET treats axioms that have no RHS or any variables in a special way. These are called *facts* , and are indicated using a simplified syntax. If a fact has an index, it is indicated as an extra argument to the predicate. In a proof, facts are always checked first before any axioms are tried. Here are a couple of facts:

```
[Father John1 Mary1]          ; The fact that John1 is the father of Mary1
[Mother Sue1 Mary1 <1].       ; The fact that Sue1 is the mother of Mary1 with
                              index <1
```

RHET automatically converts axioms with no RHS or variables into facts. Thus if you assert the axiom [[Mother [Sue1] [John1]] <1] it will be converted into the fact [Mother Sue1 John1 <1].

RHET supports lists using standard Lisp syntax. Thus a typical "member" function for lists could be defined in RHET as follows:

```
[[MyMember ?x*T-LISP (?x*T-LISP . ?y*T-LISP)]        <3 ]
[[MyMember ?x*T-LISP (?x1*T-LISP . ?y*T-LISP)]       <3     [MyMember ?x ?y]]
```

Lisp and RHET constants are distinguished in RHET.For example, *:JOHN* is a LISP atom and *[JOHN]* is a RHET constant. RHET can identify RHET objects by context. Thus *JOHN* is a RHET object when in a context signalled by RHET parentheses. Thus, [P A B] is a RHET literal consisting of a predicate name P and two RHET constants [A] and [B]. RHET also allows mixed terms. For example, ([P] :A [B]) is a list consisting of the RHET constant [P], the LISP atom :A and the RHET constant [B]. [P :A B] is a RHET literal with predicate P, and two arguments: a Lisp atom :A, and a RHET constant [B].

RHET also supports numbers in a way similar to PROLOG. RHET supports the general type *T-Number, which sub-divides into three different number types *T-Integer, *T-Float and *T-Rational, corresponding to the corresponding types in Common Lisp. Any Common Lisp numeric expression involving integers, floats and rationals can be evaluated in RHET, and bound RHET variables can be used freely in expressions. Unbound variables are not permitted in numeric expressions. Thus

```
(+ 1 2)
```

is an acceptable numeric expression, as is

```
(+ ?x*T-Integer 3)
```

if ?x*T-Integer is bound.

1.2 Special Operators in RHET

Like PROLOG, RHET offers a range of special operators that are useful for making certain operations more convenient and for introducing control commands. This section reviews the common operators that can be found in most PROLOGs. Unless explicitly mentioned, these operators are not *assertable*, i.e., they cannot be added to the database. For example, it would make no sense to add axioms and facts defining [Cut] to the database. But, of course, [Cut] can appear in axioms that are added to the database.

[And $Form_1$ ... $Form_N$**]**

This is true iff all the forms can be proven. **And** evaluates its arguments in the order supplied and short-circuits evaluation if possible. Thus any form following a form that fails will not be evaluated. The null form **[And]** automatically succeeds. If **[Cut]** is encountered as a form, backtracking past it cuts out of the entire **And**, but not necessarily the entire rule.

[Bound *Term*]

Succeeds only if the specified term is a bound variable. It fails on any other term such as an unbound variable or a constant. Thus if ?x is bound to [Foo], then [Bound ?x] will succeed, whereas [Bound ?y] and [Bound [Foo]] will fail. Note that [Bound <term>] differs from [Unless [Var <term>]] on how they treat constants.

[Cond $(TestForm\ ActForm_1 \ldots ActForm_n)^*$**)**

This expression is like the Lisp COND statement, the first testForm to succeed results in its ActForms being proven. For example, [COND ([p1] [a1] [a2]) ([p2] [a3])] is equivalent to [OR [AND [p1] [CUT] [a1] [a2]] [CUT] [AND [p2] [CUT] [a3]]].

[Cut]

The Cut literal has no effect until RHET tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on.

[Fail]

This predicate is always false.

[Ground *Term*]

This succeeds only if the term does not contain any unbound variables.

[Member *Term List*]

This predicate is true if the specified term is a member of the specified list.

[Or $Form_1$ ... $Form_N$**]**

This expression is true if any of the specified forms can be proven. The 0-arity form, **[OR]**, automatically fails. The evaluation of forms will be short-circuited if possible, i.e. it will stop at the first form that evaluates to true. Note that the **Cut** operator is not caught by this form: backtracking through a **Cut** will cause the rule containing the **Or** to fail.

[Rprint *Term1 ... TermN*]

The specified terms are pretty-printed on successive lines to the default output stream. The stream is set by setting the Common Lisp default output stream.

[Rterpri]

This print a line feed and carriage return to the default output stream

[Unless *Form1 ... FormN*]

This expression is true only if the specified forms cannot be proven. This is the standard proof-by-failure mechanism in PROLOG. Warning, RHET also has a predicate **Not**, but this is not negation as failure. It has a special interpretation as described in section 2.

[Var *Term*]

This succeeds only if the term is an unbound variable.

[Win]

This predicate always succeeds and is useful inside **Cond** forms.

RHET supports the following predicates for numeric manipulation:

[:= *Variable Expression*]

Evaluates the expression and binds the variable to the resulting value. Thus
[:= ?x*T-Integer (* 3 (+ 4 2))]
will succeed and binds ?x*T-Integer to the value 18.

[== *Expression1 Expression2*]

This is true if the evaluation of the two expressions yields the same value using the Lisp function EQL.

[=/= *Expression1 Expression2*]

This is true if the evaluation of the two expressions yields different values.

RHET also supports numeric comparison operators with the obvious interpretations.

[< *Expression1 Expression2*] - Expression1 is less than Expression2

[> *Expression1 Expression2*] - Expression1 is greater than Expression2

[<= *Expression1 Expression2*] - Expression1 is less than or equal to Expression2

[>= *Expression1 Expression2*] - Expression1 is greater than or equal to Expression2.

1.3 Using The RHET.1 Subset

Typically, the functions that are used to call RHET have arguments that are RHET forms. One argument that is very common is a called a *headPattern*. A headPattern is simply a form used to select all axioms whose head unifies with it. Thus the headpattern [P ?x Jack] would select every axiom with a head that unifies with [P ?x Jack]. You may select every axiom involving a predicate name *P* (even those with a varying number of arguments) with [P &rest ?x*T-LISP] (or [P &rest ?x] since the type of ?x will default to the right type). When a function may take several different arguments, the set of options will be listed within curly brackets. The primary function

for adding facts and axioms to RHET is the following:

(Rassert *Axiom$_1$ ... Axiom$_n$*)

This function can be used to assert arbitrary RHET axioms and facts. For example, we can define a small database for parenthood predicates as follows:

```
RHET -> (Rassert
        [[Parent ?x*T-U ?y*T-U] <INDEX1 [Father ?x ?y]]
        [[Parent ?x*T-U ?y*T-U] <INDEX2 [Mother ?x ?y]]
        [[Mother [Sue] [John]] <DATA]
        [[Father [Sam] [John]] <DATA]
        [[Mother [Sue] [Jack]] <DATA])
```

to which RHET will respond:

```
([MOTHER SUE JACK <DATA] [FATHER SAM JOHN <DATA]
        [MOTHER SUE JOHN <DATA]
        [[SBMB [PARENT ?x*T-U  ?y*T-U ]] <INDEX2 [MOTHER ?x*T-U  ?y*T-U ]]
        [[SBMB [PARENT ?x*T-U  ?y*T-U ]] <INDEX1 [FATHER ?x*T-U  ?y*T-U ]])
```

The response lists all the facts and axioms added in reverse order. The "SBMB" operator around the axioms indicates that these axioms form part of what the system (RHET) believes is mutually believed. This can be ignored for now as it relates to RHET's context mechanism to be discussed later in the tutorial. For now, all axioms will be added and proofs will be done with respect to this modality.

Once axioms and facts are defined, then proofs can be done. The primary function for invoking proofs is:

(Prove *Form*)

This invokes the theorem prover which then attempts to prove the specified form using a standard backwards chaining strategy and returns an answer with the variables bound appropriately. Thus if we call

```
RHET ->  (prove [Parent ?x*T-U [John]])
    [PARENT ?x->SAM [JOHN]]
```

Multiple queries may be made simultaneously by using the **And** operator. Thus to find if Jack and John share a parent, we might query

```
RHET ->  (prove [And [Parent ?x*T-U [John]] [Parent ?x*T-U [Jack]]])
    [AND [PARENT ?x->SUE JOHN]  [PARENT ?x->SUE JACK]].
```

If you want all the answers, or a fixed number of them then you may use the function:

(Prove-All *Form* &key *Number-of-Proofs*)

This will return all answers, or the number requested in the optional argument. Here is an example with the above database:

```
RHET ->  (prove-all [Parent ?x*T-U [John]])
    ([PARENT SUE [JOHN]] [PARENT SAM [JOHN]])
```

(Prove-All [Parent ?x*T-U [John]] :number-of-proofs 1) will return only the first answer. Note that **prove-all** will find all possible answers, but not necessarily perform all possible proofs, since many proofs might return the same answer. In particular, if the query has no variables in it, then **prove-all** is equivalent to **prove**.

Finally, there are functions for deleting axioms and facts from the database and for resetting the database to its starting point.

(Remove-All *HeadPattern*)

Removes all axioms and facts that match the specified head pattern. Thus (Remove-All [P &rest ?x]) retracts all facts and axioms for the predicate *P*, while (Remove-All [P ?x A]) retracts all facts that unify with [P ?x A] and axioms whose head unifies with [P ?x A].

(Reset-Rhet)

Clears the database back to its original state when RHET was first loaded.

The following functions are also useful in some situations.

(Remove-Facts *HeadPattern*)

Removes all facts that match the specified head pattern. Thus (Remove-Facts [P &rest ?x]) retracts all facts for the predicate *P*, while (Remove-Facts [P ?x A]) retracts all facts that unify with [P ?x A].

(Remove-B-Axioms *HeadPattern*)

Retracts all axioms with a head that matches the specified pattern, which may be a predicate name, or a form that must unify with the head. Thus, (Remove-B-Axioms [P &rest ?x]) retracts all axioms with a head with the predicate *P*, and (Remove-B-Axioms [P ?x A]) retracts all axioms with a head that unifies with [P ?x A].

(Clear-Axioms)

Removes all user-defined axioms but leaves the facts untouched.

Inspecting and Debugging

The functions in this section allow the user to inspect the database and to debug proofs. Note that often the simplest way to inspect a database in RHET is to use a text editor on the file containing your axioms, just as you would inspect the definitions of Common Lisp functions in the editor rather than pretty-printing the definition in Lisp. Nevertheless, on many occasions the following functions are useful:

(List-All *HeadPattern)*

This returns a list of all facts that unify with the head pattern specified, and all axioms whose head unifies with the head pattern. The facts are listed first followed by the axioms, reflecting the order in which RHET uses this information in a proof. Thus, starting from the database resulting from the Rassert example above

RHET -> (Rassert [Parent George Sue])

RHET -> (List-All [Parent George ?x])

returns the list

([PARENT GEORGE SUE]

[[SBMB [PARENT ?x*T-U ?y*T-U]] <INDEX1 [FATHER ?x*T-U ?y*T-U]]
[[SBMB [PARENT ?x*T-U ?y*T-U]] <INDEX2 [MOTHER ?x*T-U ?y*T-U]]).

(List-Facts *HeadPattern*)

This returns all *facts* that unify with the head pattern specified. Thus, with the database resulting from the Rassert example above,

RHET -> (List-facts [Mother [sue] ?x])

returns the list

([MOTHER SUE JACK <DATA] [MOTHER SUE JOHN <DATA]).

(List-B-Axioms *HeadPattern*)

This returns a list of all *axioms* whose head unifies with the specified pattern. Thus, with the database resulting from the Rassert example above,

RHET -> (List-B-Axioms [Parent &rest ?x])

returns the list

([[SBMB [PARENT ?x*T-U ?y*T-U]] <INDEX1 [FATHER ?x*T-U ?y*T-U]]
[[SBMB [PARENT ?x*T-U ?y*T-U]] <INDEX2 [MOTHER ?x*T-U ?y*T-U]])

(List-Fact-References *Form**)

This returns all facts that have all the specified forms as arguments. This can be used to find all the facts that mention a certain object. For instance (List-Fact-References [Sue]) will return all facts that have [Sue] as an argument.

(List-Facts-with-Bindings *HeadPattern*)

Same as **List-Facts** except that it returns the variable bindings as well in the format ((<fact> <binding list>)*). This is useful in some situations where you are interested in a particular variable's values rather than the actual facts. Thus, with the database resulting from the Rassert example above,

RHET -> (List-facts-with-bindings [Mother [sue] ?x])

returns the list

(([Mother Sue Jack <DATA] (?x [Jack])) ([Parent Sue John <DATA] (?x [John])).

The RHET trace facility gives the user the ability to trace predicates, set breakpoints and to single-step through proofs. This can all be invoked through RHET's menu-driven interface, and is also available in the following LISP functions:

(Trace-B-Axiom &rest {{*bc-axiom headpattern*} ({*bc-axiom headpattern*} *keyword**)}*

This function starts a trace on the specified predicate that will print out a trace message each time the predicate is called and returns. There are additional optional specifications to the trace command that allow you to selectively trace calls and returns, to set break points and invoke the axiom stepper. For example,

(Trace-B-Axiom [Parent &rest ?x])

will print a message each time a the system tries to prove a literal with predicate *Parent* - both when the proof is initiated (the *call*) and when the proof completes successfully or unsuccessfully (the *return*). If no argument is given, then it returns a list of all predicates that are currently traced. One can also custom tailor the tracing information given along the lines of the regular trace function in common lisp. For example, the call

(Trace-B-Axiom '([Parent ?x ?y] :Call :Break :Return :Trace))

will interrupt the proof and enter the debugger whenever the system tries to prove a literal that unifies with [Parent ?x ?y] and will print a trace message whenever the proof returns. The full set of options is as follows: actions may be specified at four points: when the predicate is called (**:Call**), when the prover calls terms on the RHS (**:Next**), when a proof succeeds or fails (**:Return**), or when terms on the RHS fail (**:Retry**). There are also four actions that may be specified: print a diagnostic message (**:Trace**), invoke a break loop (**:Break**), invoke the debugger (**:Debug**), or invoke the stepper (**:Step**).
Multiple functions may be traced in one call (and options may be specified for each if enclosed in parentheses). Thus to do the above trace and to also trace the predicate Mother, we could call

```
(Trace-B-Axiom    '([Parent ?x ?y] :Call :Break :Return :Trace)
                  [Mother &rest ?x]).
```

For example, consider tracing a proof given the axioms asserted above, where the trace output is in the smaller font size.

```
RHET -> (Trace-B-Axiom [parent ?x ?y])
    ([[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX2 [MOTHER ?X*T-U  ?Y*T-U ]]
    [[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX1 [FATHER ?X*T-U  ?Y*T-U ]])
RHET -> (prove-all [parent ?x ?y])
    Matched and Interpreting Axiom [[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX1  [FATHER ?X*T-U
        ?Y*T-U ]]
    Axiom Succeeds ([[SBMB [PARENT ?X->SAM ?Y->JOHN]] <INDEX1
        [FATHER ?X->SAM ?Y->JOHN]])
    Axiom Fails ([[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX1 [FATHER ?X*T-U  ?Y*T-U ]])
    Matched and Interpreting Axiom [[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX2
        [MOTHER ?X*T-U  ?Y*T-U ]]
    Axiom Succeeds ([[SBMB [PARENT ?X->SUE ?Y->JOHN]] <INDEX2
        [MOTHER ?X->SUE ?Y->JOHN]])
    Axiom Succeeds ([[SBMB [PARENT ?X->SUE ?Y->JACK]] <INDEX2
        [MOTHER ?X->SUE ?Y->JACK]])
    Axiom Fails ([[SBMB [PARENT ?X*T-U  ?Y*T-U ]] <INDEX2 [MOTHER ?X*T-U  ?Y*T-U ]])
    ([PARENT SUE JACK] [PARENT SUE JOHN] [PARENT SAM JOHN])
```

(UnTrace-B-Axiom &Optional *{BC-Axiom PredName}*)

This function simply turns off tracing for the predicate specified. The form of the argument is specified in Trace-B-Axiom. If no argument is specified, all predicates will be untraced.

(Trace-Builtins)

Builtin predicates cannot be traced individually, but can't be traced as a group. This function causes a detailed tracing of all builtins and user defined predicates.

(Untrace-Builtins)

This turns off full tracing.

(Rhet-Dribble-Start *File-Spec* &Optional (Mode :Both))

Starts a dribble file. The File-Spec is a string that specifies a file in the format used by the host Operating System.

(Rhet-Dribble-End)

Ends dribbling and properly closes the file.

Using Indices

The index mechanism in RHET provides an alternate way to organize axioms for inspection, retrieval and deletion. All these functions take an argument that is an *index pattern*. The simplest index pattern is an atom. When an atom is used, all axioms and/or facts with exactly that atom as an index are affected. RHET also supports a regular expression language for building complex patterns to match against indices. The exact format is specified in the reference manual.

(List-All-By-Index *indexExpression*)

This returns all facts and axioms that match the index pattern specified. If the index is an atom, this is simply the facts and axioms with that atom as an index.

(List-Facts-By-Index *indexExpression*)

This returns all facts that match the index pattern specified. If the index is an atom, this is simply the facts with that atom as an index. An example with the above database is

RHET -> (List-Facts-by-index '<DATA)

([MOTHER SUE JOHN <DATA] [FATHER SAM JOHN <DATA]
[MOTHER SUE JACK <DATA])

(List-B-Axioms-By-Index *indexExpression*)

This returns a list of all axioms that match the index pattern specified. For example

RHET -> (list-b-axioms-by-index '<INDEX1)

([[SBMB [PARENT ?x*T-U ?y*T-U]] <index1 [FATHER ?x*T-U ?y*T-U]])

(Remove-All-by-Index *indexExpression*)

Retracts all facts and axioms with indices that match the index pattern specified

(Remove-Facts-by-Index *indexExpression*)

Retracts all facts with indices that match the index pattern specified. An example is

RHET -> (Remove-facts-by-index '<DATA)

(Remove-B-Axioms-By-Index *indexExpression*)

Removes all axioms with indices that match the index pattern specified. An example is

RHET -> (remove-b-axioms-by-index '<INDEX1).

1.4 Examples

This is an example adapted from Clocksin and Mellish [1981] concerning a database of library facilities.

```
RHET-> (Rassert
    ;;; Facilities that people with overdue books can have.
        ;;; we use [cut] here since, we are interested in whether he has overdue books or not
        ;;; and not how many overdue books he has.
    [[Facility ?person*T-U ?fac*T-U]        <FAC
        [BookOverDue ?person ?book*T-U] [cut] [BasicFacility ?fac]]
    ;;; this is the facilities general people can have.
    [[Facility ?person*T-U ?fac*T-U]        <FAC [GeneralFacility ?fac]]
    ;;; these are the basic facilities for everyone.
    [[BasicFacility  [Reference]]           <BASIC ]
    [[BasicFacility  [Enquiries]]           <BASIC ]
    ;;; these facilities are kept away from people with overdue books
    [[AdditionalFacility [Borrowing]]       <ADD ]
    [[AdditionalFacility [InterLibraryLoan]] <ADD ]
    ;;; these are all facilities a person without overdue books can enjoy.
    [[GeneralFacility ?fac*T-U]             <GEN [BasicFacility ?fac]]
    [[GeneralFacility ?fac*T-U]             <GEN [AdditionalFacility ?fac]]
    ;;; facts that indicate which person has which overdue books
    [[BookOverDue [John] [Rhet.manual]]     <OVERDUE ]
    [[BookOverDue [John] [Common.lisp]]     <OVERDUE ]
    ;;; legal users of the library
    [[Client [John]]                        <CLIENT ]
    [[Client [Mary]]                        <CLIENT ])
    ([CLIENT MARY <CLIENT] [CLIENT JOHN <CLIENT]
        [BOOKOVERDUE JOHN COMMON.LISP <OVERDUE]
        [BOOKOVERDUE JOHN RHET.MANUAL <OVERDUE]
        [[SBMB [GENERALFACILITY ?fac*T-U ]] <GEN
            [ADDITIONALFACILITY ?fac*T-U ]] [
        [SBMB [GENERALFACILITY ?fac*T-U]] <GEN
            [BASICFACILITY ?fac*T-U ]]
        [ADDITIONALFACILITY INTERLIBRARYLOAN <ADD]
        [ADDITIONALFACILITY BORROWING <ADD]
        [BASICFACILITY ENQUIRIES <BASIC] [
        BASICFACILITY REFERENCE <BASIC]
        [[SBMB [FACILITY ?person*T-U  ?fac*T-U ]] <FAC
            [GENERALFACILITY ?fac*T-U ]]
```

```
[[SBMB [FACILITY ?person*T-U  ?fac*T-U ]] <FAC
   [BOOKOVERDUE ?person*T-U  ?book*T-U ]
   [CUT]
   [BASICFACILITY ?fac*T-U ]])
; Testing the data
;;; see what facilities are open to John, who has overdue books
RHET -> (prove-all [Facility [John] ?fac*T-U])
      ([FACILITY [JOHN] ENQUIRIES] [FACILITY [JOHN] REFERENCE])
;;; see what facilities are open to Mary.
RHET -> (prove-all [Facility [Mary] ?fac*T-U])
      ([FACILITY [MARY] INTERLIBRARYLOAN] [FACILITY [MARY] BORROWING]
            [FACILITY [MARY] ENQUIRIES] [FACILITY [MARY] REFERENCE])
;;; the overdue books information is grouped under the same index <OVERDUE
RHET -> (List-Facts-by-index '<overdue)
      ([BOOKOVERDUE JOHN RHET.MANUAL <OVERDUE]
            [BOOKOVERDUE JOHN COMMON.LISP <OVERDUE])
;;; assume that Mary just has a overdue book, we can just insert this into the database.
RHET -> (Rassert [[BookOverDue [mary] [Whatever.book]] <OVERDUE])
      ([BOOKOVERDUE MARY WHATEVER.BOOK <OVERDUE])
;;; then the facilities open to her is restricted.
RHET -> (prove-all [Facility [Mary] ?Fac*T-U])
      ([FACILITY [MARY] ENQUIRIES] [FACILITY [MARY] REFERENCE])
;;; we can still extract all the overdue books using the index.
RHET -> (List-Facts-by-index '<overdue)
      ([BOOKOVERDUE JOHN RHET.MANUAL <OVERDUE]
            [BOOKOVERDUE JOHN COMMON.LISP <OVERDUE]
            [BOOKOVERDUE MARY WHATEVER.BOOK <OVERDUE])
```

1.5 Other Useful Functions and Predicates

This section describes some additional functions and builtin predicates that are useful in many applications. In addition, section 5 (the RHET.5 subset) contains additional functions relating to the integration of RHET and Common Lisp. You can skip directly to section 5 if you need these facilities.

The following builtin predicates are mostly procedural in nature and often have side effects. Unless explicitly noted otherwise, these builtins are not assertable.

[Assert-Axioms $Axiom_1$ ... $Axiom_N$**]**

This asserts the specified axioms or facts into the database just as the Lisp function **Rassert**. All variables in the axioms that also appear outside the axiom assertion will be replaced by their binding before the axiom is added. Thus in an environment where ?x is bound to [Foo], the term

[Assert-Axioms [[P ?x ?y] <1 [Q ?x ?y]]]

will add the axiom

[[P [Foo] ?y] <1 [Q [Foo] ?y]].

[Assert-Fact *Fact$_1$ ... Fact$_N$*]

This adds the specified facts into the database. All variables appearing in the facts to be added must be bound. An unbound variable will generate an runtime error.

[Bagof *Var1 Form Var2*]

This is a faster version of **SetAll** that doesn't check for duplications in the list bound to Var1. It is true if *Var1* is set to a list of all assignments to *Var2* that satisfy the specified form. See **Setall** for examples.

[Forall! *vars defForm testForm$_1$ ... testForm$_n$*]

This expression is true if for every binding of a variable in **vars**, if **defForm** succeeds, then all **testForm$_i$**'s succeed. Note that **vars** may be a single variable or a list of variables, and all variables should appear in **defForm** if this is to do what you expect! Thus [FORALL! ?x [P ?x] [Q ?x]] is true only if every binding of ?x satisfying [P ?x] also satisfies [Q ?x]. This predicate can be used in assertions. If [Forall! ?x [P ?x] [Q ?x]] is asserted, then RHET asserts [Q ?x] for every ?x such that it can prove [P ?x]. It can also

[Retract *Form*]

This retracts all facts (not axioms) that unify with the specified form.

[RFormat *Stream control-String &rest Form**]

This is modelled after the Common Lisp **Format** function. The forms specified are printed according to the **control-String** on the specified **Stream**. See a Common Lisp manual for details. Normally the stream will be :T for the default, but it can also be a Lisp expression that will evaluate to a stream.

[SetAll *Setvar Form Var*]

This expression is true if the specified setvar is the list of all bindings of the specified var that can make the specified form equivalent to an asserted fact. This function does *not* invoke backward chaining, it only queries the existing database.[1] Thus if the database contains [P A B], [P C D], and [P E D], then [SetAll ?x*T-List [P ?y ?z] ?z] will be provable with ?x bound to ([B] [D]).

2. RHET.2: RHET as a Deductive Data Base

The RHET.2 subset extends the PROLOG base of RHET.1 with several capabilities that make it useful for many knowledge representation applications. These extensions are forward chaining axioms, a capability to postpone evaluation of expressions until all variables in it are bound, and a limited capability for handling negation.

To distinguish them from forward chaining axioms, we will call the PROLOG-style axioms introduced in section 1 backward chaining axioms. Forward and backward chaining axioms have

---

[1] If you want to invoke backward chaining, this can be done using **Forall!**, or using **Genvalue** documented in section 5.

essentially the same syntax, the only difference is how they are used by the system. Backward changing axioms are applied when the system is attempting prove some set of formulae. Forward chaining axioms are used to perform inference when new facts are added to the data base. These correspond exactly to the consequent and antecedent theorems introduced in PLANNER. A forward chaining axiom has one or more triggers, which are literals that indicate when the axiom is to be used, namely whenever a fact is added that matches one of the triggers. The general form of a forward chaining axiom is

[Head Index Literal$_1$ ... Literal$_n$:forward Trigger$_1$ ... Trigger$_m$].

When a forward chaining axiom's trigger is matched, the variable bindings are used to instantiate the variables in the axiom. Then if all the literals on the right-hand side of the axiom can be successfully unified with facts in the data base, the head is added to the database. Note that there is no attempt to prove the literals on the right-hand side, they are simply matched into the database. Adding the head may then recursively invoke additional forward chaining axioms.

Consider an example: Let us define a predicate Above to be the transitive closure of a predicate On. The following two forward chaining axioms could be used:

Head	Right Hand Side		Trigger
[[Above ?x ?y] <1	[On ?x ?y]	:forward	[On ?x ?y]]
[[Above ?x ?y]<2	[On ?x ?z] [Above ?z ?y]	:forward	[Above ?z ?y]]

If we add [On A B], then the trigger of axiom 1 matches. Thus axiom 1 is attempted. The right hand side is simply the trigger itself, so the axiom applies and [Above A B] is added to the database. This time, the trigger for axiom 2 matches. In attempting axiom 2, however, no literal of the form [On ?x A] is found in the database, so the axiom does not apply. If [On B C] is now added, then the trigger for axiom 1 matches, and the axiom applies resulting in [Above B C] being added to the data base. Now the trigger on axiom 2 matches. In this case, the right hand side can be found in the database since [Above B C] and [On A B] are in the database, so [Above A C] is added to the database.

Forward chaining occurs only when a fact matching a trigger is added. In particular, no forward chaining is attempted when a forward chaining axiom is added, even if all the literals on the right hand side already exist in the database.

RHET uses truth maintenance techniques to keep track of facts that are added by forward chaining. This information is used in retraction. If a fact is retracted, then all conclusions that were derived by forward chaining from this fact are retracted as well unless they have independent support (i.e., it could be inferred by another forward chaining axiom that is still valid, or it was directly asserted by the user). This technique is limited however, and cannot handle changes that are made using builtin functions that add or delete facts from the database. For instance, if adding fact P caused some other fact Q to be retracted from the database, then retracting P would not restore Q.

One of the problems with forward chaining systems is that if an inconsistent fact is added, the system may add a considerable number of new facts before the contradiction is detected. Since RHET maintains the justification for each newly added fact, once it finds a contradiction, it can retract all the consequences and the addition fails. RHET also supplies a capability for the user to check for and signal contradictions that the system couldn't detect.

Constraint Posting

Constraint Posting is the second new feature. This allows the user to delay the proof of certain literals until all the variables in it are bound. This is called *posting* a literal. If a literal is posted that contains no unbound variables, then the literal becomes a goal as in a normal proof. If it contains any unbound variables, however, then its evaluation is delayed until the variables are bound. The mechanism for doing this involves using a new kind of term called a constrained variable. A constrained variable is of the form

[any VarName Literal$_0$ ... Literal$_n$].

Such a variable cannot be bound to a term *t* unless all the literals that are now fully grounded once *t* is substituted for *VarName* can be proven. Thus the term

[Any ?x [P ?x]]

unifies with a ground term t only if [P t] is provable. Constrained variables can involve complex constraints using more than one variable. For instance the two variables [any ?x [P ?x ?y]] and [any ?y [P ?x ?y]] can only be bound, respectively, to terms t1 and t2 if [P t1 t2] is provable. Since the variables may not be bound simultaneously, the first one to be bound would further restrict the possible bindings for the second. For instance, the literal [Q [any ?x [P ?x ?y]] [any ?y [P ?x ?y]]] could unify with [Q A ?z] to produce [Q A [any ?y [P A ?y]]. This could then unify with [Q A B] only if [P A B] was provable.

Logically, a literal containing a constrained variable is equivalent to a conditional formula in FOPC. For example, asserting the literal [P [any ?x [R ?x]] would correspond to asserting the formula

$\forall x.\ R(x) \supset P(x).$

This correspondence can be used to explain the behavior of constrained variables during proofs. For example, [P A] is provable from [P [any ?x [R ?x]] only if [R A] is provable. This corresponds to the logical relation that $P(A)$ follows from $\forall x.\ R(x) \supset P(x)$ only if $R(A)$ is true.

So, while adding constrained variables does not expand the expressive power of the Horn clause formalism, it provides the system and users with a powerful technique for reordering the search space for proofs that can result in considerable efficiency gains. This capability will be very useful later when adding the specialized reasoning systems to RHET. It also allows RHET to return answers that might not be obtainable using a standard PROLOG strategy. For instance, RHET could return an answer such as [P [any ?x [Q ?x]] where PROLOG would only be able to return a list of answers of form [P α] where [Q α] is provable in the database.

There are several ways that the system could handle the unification of two constrained variables. It could simply allow any unification and create a new variable with the union of the constraints from the original variables. Thus [any ?x [P ?x]] and [any ?y [Q ?y]] would unify to a value [any ?z [P ?z] [Q ?z]]. The problem is that such a solution might be vacuous as there might be no value that could simultaneously satisfy both constraints. For example, [any ?x [P ?x]] and [any ?y [Not [P ?y]]] would unify to produce [any ?z [P ?z] [Not [P ?z]]]. On the other hand, it is not possible for RHET to guarantee that an arbitrary set of constraints is consistent, so RHET uses a heuristic strategy: if the database contains a term that satisfies all the constraints, the unification is allowed. Otherwise it fails.

Negation

The final extension is a limited negation facility. RHET allows literals of the form [Not *Literal*]. This is only related to its corresponding positive literal in a limited manner, but the facility is still quite useful. Negative literals can be used anywhere in axioms and the proof procedures operate as usual. For example, one could add an axiom

[[Not [Human ?x]] < [Not [Mammal ?x]]]

as the "contrapositive" of *All humans are mammals*. If we then assert [Not [Mammal George]] then RHET can prove [Not [Human George]]. Of course, nothing prevents RHET from being able to prove [Human George] from other axioms in the same database unless RHET provides some consistency checking. RHET offers only a very limited consistency check: a literal cannot be added if its negation is already present in the database. Note that many predicates that have special interpretations do not support a notion of negation. This includes the predicates that are procedural in nature (e.g. **Cut**, **Rprint**, and so on), as well as predicates that interact with the specialized reasoning systems such as equality, the type system, and so on. In general, **not** simply fails it it modified one of these predicates.

So while RHET does not allow what we might call blatant contradiction, it is simple to add axioms so that a literal and its negative form are both provable. RHET offers the user several proof modes in order to deal with such situations. In *simple* proof mode, RHET simply attempts to prove the literal given as the goal, just as in standard PROLOG. In *question answering* (or *default*) mode, RHET attempts to prove both the literal and its negation using the simple PROLOG strategy. Depending on the result of the two proofs, there are four possible answers to a proof of some literal [P]:

:UNKNOWN - neither P nor [not P] are provable,
[P] (i.e. true) -- P is provable and [not P] is not,
NIL (i.e. false) - [not P] is provable and P is not, and
:INCONSISTENT - both P and [not P] are provable.

The final mode is *Complete* reasoning mode. This is like question answering mode, except that every subgoal also invokes two subproofs: one for its positive version and one for its negative. Whenever it finds an inconsistency, it simply ignores that predication for the rest of the proof. Thus, if there is some other way to prove the formula not involving the inconsistency, then the answer might return true. If there is no other way to prove the formula, the answer would be :UNKNOWN.This last version can be very expensive but guarantees that a goal is not proven using any provably inconsistent assertions. As an example, consider what can be proved in the different modes given the following database of axioms:

[P], [[Q] < [P]], [[not Q] < [P]], [not R], [[S] < [Q]].

Figure 1 indicates the results of trying to prove [P], [Q], [R]. [S] and [T] in the three different proof modes. When trying to prove [S] in complete mode, RHET enters the debugger when the inconsistency based on [Q] is found. If you allow the proof to continue from this point, it will return :UNKNOWN since there is no other way to prove [S] or [not S].

Goal	:simple mode	:default mode	:complete mode
[P]	[P]	[P]	[P]
[R]	NIL	NIL	NIL
[Q]	[Q]	:INCONSISTENT	:INCONSISTENT
[S]	[S]	[S]	Debugger
[T]	:UNKNOWN	:UNKNOWN	:UNKNOWN

Figure 1: The results of proofs in the three proof modes

2.1 Syntax

As seen above, forward chaining axioms are identical in form to backwards chaining axioms except for the keyword and the specification of the triggers. Repeating the definition from above, the general form is

[Head Index Literal_1 ... Literal_n :forward Trigger_1 ... Trigger_m].

If no trigger is specified, the every literal on the right hand side of the axiom acts as a trigger.

When a forward chaining axiom is applied, the literals on right hand side are only checked by unification into the database. In particular, the backwards chaining prover is not invoked while using a forward chaining axiom. For example, if we had the forward chaining axioms above, plus a backwards chaining axiom such as

[[On ?x ?y] < [TopTouchesBottom ?x ?y]]

and the literal [TopTouchesBottom A B] was in the database. Then if we added [On B C] to the database, axiom 1 would apply and add [Above B C]. But axiom 2 would not apply since [On A B] is not in the database, even though it is provable. In order to allow the backwards chaining prover to be invoked while forward chaining, a builtin predicate is provided that corresponds to the previously described LISP function **prove**:

[Prove *Form*]

Succeeds if Form can be proven using backwards chaining. Any variables bound in the query form to make the proof succeed will remain bound.

For example, if we have the backward chaining axioms above and following forward chaining axioms:

Head	*Right Hand Side*		*Trigger*
[[Above ?x ?y] <1	[On ?x ?y]	:forward	[On ?x ?y]]
[[Above ?x ?y] <2	[Prove [On ?x ?z]] [Above ?z ?y]	:forward	[Above ?z ?y]]

then adding [On B C] to a database containing [TopTouchesBottom A B] would result in both [Above B C] and [Above A B] being added to the database.

Constrained variables are introduced using the special form:

[Post *Literal*]

If the literal is fully grounded, then this succeeds only if the literal is provable. Otherwise, it succeeds and all unbound variables are bound to any forms that encode the constraint indicated by the literal.

For example, consider using the clause

[FindHappyEmployee ?x ?y] < [Post [Employs ?x ?y]] [Happy ?y]

to prove the goal

[FindHappyEmployee Jack ?y].

The first subgoal [Post [Employs Jack ?y]] succeeds and binds ?y to [any ?z [Employs Jack ?z]. Thus the second subgoal is [Happy [any ?z [Employs Jack ?z]]], which will be provable if there is an assertion [Happy t] in the database such that [Employs Jack t] is provable.

One useful builtin predicate uses constraint posting. It succeeds if two terms are distinct and works even if the terms are originally unbound variables:

[Distinct *Term1 Term2*]

True only if Term1 and Term2 are not equal. If either term (or both) are variable(s), this check is delayed until the variable(s) are bound. [Distinct ?x ?y] is functionally equivalent to [Post [Unless [EQ? ?x ?y]]]. The predicate is not assertable.

2.2 Using the RHET.2 Subset

All the functions defined in the RHET.1 subset still apply, often with additional optional arguments to handle the extensions. For example, the function **prove** is extended with a keyword to indicate the mode of proof.

(Prove *Form* &key *Mode*)

Attempts to prove the specified form using the mode indicated. The default mode is question-answering mode, **:simple** indicates simple proof mode (i.e. the normal prolog strategy) and **:complete** indicates complete mode.

Assert is extended to handle forward chaining axioms. For example, the above forward chaining example would be asserted as follows:

```
(Rassert
    [[Above ?x ?y] <1 [On ?x ?y] :forward]
    [[Above ?x ?y] <2 [On ?x ?z] [Above ?z ?y]  :forward [Above ?z ?y]]
```

The builtin predicate Distinct defined above could be defined as follows:

(Rassert [[MyDistinct ?x ?y] <3 [Post [Unless [EQ? ?x ?y]]]]).

Note also that since we can now assert the negation of some fact, it may be that a fact cannot be added to the database because its negation is already in the database, and thus adding the fact would make the database inconsistent. **Rassert** is defined so that the contradictory information is retracted and the new facts are added.

The RHET predicates dealing with asserting facts handle inconsistencies in different ways. For example, if one uses **Assert-Axioms** or **Assert-Fact** to add a fact, then before RHET adds the fact, it tries to prove its negation. If its negation can be proved, then RHET enters the debugger. The builtin predicate **Assert-if-Consistent**, on the other hand, only adds a fact if it is consistent, and otherwise quietly fails. To summarize, we have the following:

[Assert-Axioms *Axiom$_1$ ... Axiom$_N$*]

[Assert-Fact *Fact$_1$ ... Fact$_N$*]

These add the specified axioms into the database just as specified in section 1. If the negation of one of the facts already exists in the database, however, then RHET enters the debugger. If you do not want this behavior, you should either use **Assert-if-Consistent** below, or explicitly test for consistency before adding a fact.

[Assert-if-Consistent *Axiom$_1$ ... Axiom$_N$*]

Succeeds and adds the specified axioms (or facts) if none of them cause a detectable inconsistency when added. If an inconsistency is detected, none are added and the predicate fails.

Because of the limitations of Horn-clause proof strategies, not all inconsistencies can be detected by RHET. But all simple inconsistencies between facts (e.g. trying to add *p* and [Not *p*] for any fact *p* is always detected).

Since a predicate may have both forward and backwards axioms defined for it, RHET.2 introduces a set of functions for manipulating axioms and tracing that correspond to those in RHET.1 for backwards chaining. These are:

(Remove-F-Axioms *TriggerPattern*)

Retracts all forward chaining axioms whose trigger matches the specified pattern.

(Remove-F-Axioms-By-Index *indexExpression*)

Retracts all forward chaining axioms with indices that match the index pattern specified

(List-F-Axioms *TriggerPattern*)

Lists all forward chaining axioms whose trigger matches the specified pattern. (List-F-Axioms [P &rest ?x]) would list all forward chaining axioms with a trigger with a predicate name P, while (List-F-Axioms [P A ?y]) would list all forward chaining axioms whose trigger matches [P A ?y].

(List-F-Axioms-By-Index *indexExpression*)

This returns a list of all forward chaining axioms that match the specified index pattern.

(Trace-F-Axiom *{FC-axiom TriggerPattern} Keyword**)

Traces every forward chaining axiom whose trigger unifies with the specified pattern. As with **Trace-B-Axiom**, there are different options for setting trace points and different actions that can be specified. The default setting is to trace the :call and :return points by printing a message at the terminal (the :trace option). The other options are as documented for **Trace-B-Axiom** in section or in the reference manual. This function can also be used to set tracing several predicates in one call as with **Trace-B-Axiom.** If no predicate is specified, then it returns a list of all forward chaining axioms currently being traced.

(List-Forward-Chained-Facts)

This function returns all the assertions made by forward chaining since the last time this function was called.

(UnTrace-F-Axiom &Optional *Form*)

This function turns off the tracing of forward chaining for any predicate matching the specified Form.

The default reasoning mode for RHET is the question-answering strategy, i.e. with each query P it attempts to prove [P] and [Not P]. This can be changed using the function:

(Set-Reasoning-Mode *{:Simple :Default :Complete}*)

This sets the reasoning mode of all proofs (except those that explicitly override the default by an argument to **prove**) to the indicated mode.
:Simple - standard PROLOG proof strategy
:Default - the question answering mode - tries to prove the goal and its negation
:Complete - checks every subgoal for inconsistency

2.3 Examples

Here are some examples using the new features.

```
; A person will buy something if it is desirable, they can afford it and they don't own
;    it already. Let's post the desirable predicate for the demo. This might be useful
;         if it were very expensive to prove the desirable predicate.
RHET -> (Rassert
                [[CouldBuy ?p ?i] < [Desirable ?i] [CanAfford ?p ?i] [Not [Own ?p ?i]]]
                ; A person can afford something if they have enough money
                [[CanAfford ?p ?i] <
                    [wealth ?p ?w*T-Integer]
                    [Cost ?i ?price*T-Integer]
                    [> ?w ?price]]
                ; a forward chaining axiom: if an object costs over $100, then it is desirable
                 [[desirable ?x ] <
                    [cost ?x ?amt*t-integer]
                    [> ?amt 100]
                    :forward  [cost ?x ?amt*t-integer]])
        ((([[DESIRABLE ?X] < [COST ?X ?AMT*T-INTEGER ] [> ?AMT*T-INTEGER  100]
              :forward [COST ?X ?AMT*T-INTEGER ]])
           [[SBMB [CANAFFORD ?P ?I]] < [WEALTH ?P ?W*T-INTEGER ]
              [COST ?I ?PRICE*T-INTEGER]  [> ?W*T-INTEGER ?PRICE*T-INTEGER]]
           [[SBMB [COULDBUY ?P ?I]] < [DESIRABLE ?I] [CANAFFORD ?P ?I]
              [NOT [OWN ?P ?I]]])
; turn on forward chaining tracing on the cost predicate
RHET ->(Trace-F-Axiom [Cost &rest ?x])
        ([[DESIRABLE ?X] < [COST ?X ?AMT*T-INTEGER ] [> ?AMT*T-INTEGER  100]
              :forward [COST ?X ?AMT*T-INTEGER ]])
; now we add some prices of objects
RHET -> (Rassert [Cost A1 150] [Cost A2 150] [Cost A3 200]  [Cost A4 500]
```

```
                [Cost A5 75] [Cost A6 50])
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A1] < [COST ?X->A1 ?AMT->150] [> ?AMT->150 100]
                    :forward [COST ?X->A1 ?AMT->150]]
                >>>Added [DESIRABLE A1 <]
                 With Justifications:  [COST A1 150 <]
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A2] < [COST ?X->A2 ?AMT->150] [> ?AMT->150 100]
                    :forward [COST ?X->A2 ?AMT->150]]
                >>>Added [DESIRABLE A2 <]
                 With Justifications:  [COST A2 150 <]
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A3] < [COST ?X->A3 ?AMT->200] [> ?AMT->200 100]
                    :forward [COST ?X->A3 ?AMT->200]]
                >>>Added [DESIRABLE A3 <]
                 With Justifications:  [COST A3 200 <]
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A4] < [COST ?X->A4 ?AMT->500] [> ?AMT->500 100]
                    :forward [COST ?X->A4 ?AMT->500]]
                >>>Added [DESIRABLE A4 <]
                 With Justifications:  [COST A4 500 <]
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A5] < [COST ?X->A5 ?AMT->75] [> ?AMT->75 100]
                    :forward [COST ?X->A5 ?AMT->75]]
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A6] < [COST ?X->A6 ?AMT->50] [> ?AMT->50 100]
                    :forward [COST ?X->A6 ?AMT->50]]
        ([COST A6 50 <] [COST A5 75 <] [COST A4 500 <] [COST A3 200 <] [COST A2 150
                <] [COST A1 150 <])
; Lets inspect some of the database - here are some facts added by forward chaining
RHET -> (List-Facts [Desirable ?x])
        (([DESIRABLE A4 <] [DESIRABLE A3 <] [DESIRABLE A2 <] [DESIRABLE A1 <]))
;  If we retract [Cost A1 150], then [Desirable A1] is also retracted
RHET-> (Remove-All [Cost A1 ?x])
        NIL
RHET -> (List-Facts [Desirable ?x])
        (([DESIRABLE A4 <] [DESIRABLE A3 <] [DESIRABLE A2 <]))
; Lets add it back
RHET -> (Rassert [Cost A1 150])
                >FC:>Matched and Interpreting Axiom
                    [[DESIRABLE ?X->A1] < [COST ?X->A1 ?AMT->150] [> ?AMT->150 100]
                    :forward [COST ?X->A1 ?AMT->150]]
        ([COST A1 150 <])
```

```
; Here are some facts about wealth and ownership
RHET -> (Rassert
            ; Jack has $225
            [wealth Jack 225]
            ; Jack already owns A1, doesn't own A3 or A4, and we don't know whether Jack
            ; owns A2, A4 or A6
            [Own Jack A1] [Not [Own Jack A3]] [Not [Own Jack A4]])
      (NOT[OWN JACK A4 <] NOT[OWN JACK A3 <] [OWN JACK A1 <]
            [WEALTH JACK 225 <])
; Turning on tracing
RHET -> (Trace-B-Axiom [CanAfford &rest ?x])
      ([[SBMB [CANAFFORD ?P ?I]] <
            [WEALTH ?P ?W*T-INTEGER ]
            [COST ?I ?PRICE*T-INTEGER ]
            [ ?W*T-INTEGER ?PRICE*T-INTEGER ]])
; Lets find out what we can prove that Jack could buy
RHET -> (Prove-All [CouldBuy Jack ?i])
            Matched and Interpreting Axiom
            [[SBMB [CANAFFORD ?P->JACK ?I->A2]] < [WEALTH ?P->JACK ?W*T-INTEGER ]
                  [COST ?I->A2 ?PRICE*T-INTEGER ] [> ?W*T-INTEGER ?PRICE*T-INTEGER]]
            Axiom Succeeds
            ([[SBMB [CANAFFORD ?P->JACK ?I->A2]] < [WEALTH ?P->JACK ?W->225]
                  [COST ?I->A2 ?PRICE->150] [> ?W->225 ?PRICE->150]]) : NIL
            Matched and Interpreting Axiom
            [[SBMB [CANAFFORD ?P->JACK ?I->A3]] < [WEALTH ?P->JACK ?W*T-INTEGER ]
                  [COST ?I->A3 ?PRICE*T-INTEGER ] [> ?W*T-INTEGER ?PRICE*T-INTEGER]]
            Axiom Succeeds
            ([[SBMB [CANAFFORD ?P->JACK ?I->A3]] < [WEALTH ?P->JACK ?W->225]
                  [COST ?I->A3 ?PRICE->200] [> ?W->225 ?PRICE->200]]) : NIL
            Matched and Interpreting Axiom
            [[SBMB [CANAFFORD ?P->JACK ?I->A4]] < [WEALTH ?P->JACK ?W*T-INTEGER ]
                  [COST ?I->A4 ?PRICE*T-INTEGER ] [> ?W*T-INTEGER ?PRICE*T-INTEGER]]
            Axiom Fails
            ([[SBMB [CANAFFORD ?P->JACK ?I->A4]] < [WEALTH ?P->JACK ?W*T-INTEGER ]
                  [COST ?I->A4 ?PRICE*T-INTEGER] [> ?W*T-INTEGER ?PRICE*T-INTEGER]])
      ([COULDBUY JACK A3])
; Note that A3 is the only answer. RHET also can prove that A4 is not owned by Jack
            ; but Jack doesn't have enough money to buy A4. RHET cannot prove that Jack
            ; doesn't own any of the other objects
; Finally, here's a simple example showing the use of constrained variables in queries.
RHET -> (rassert [P a] [Q a])
      ([Q A <] [P A <])
RHET -> (prove [P [any ?x [Q ?x]]])
```

[P ?X->A]

2.4 Procedural Uses of Forward Chaining

The forward chaining system can be used to more procedural effect in managing the database by using builtins on the left hand side of the forward chaining axiom. While this can be done, there is no truth maintenance for such uses. Thus, retracting the effect of axioms that involve **Retract** and other builtins must be done by the programmer. Here is an example of a simple axioms that update the database to keep track of the wealth and ownership of items when a **buy** action is performed. The database starts in the state defined in the preceding section.

```
RHET -> (Rassert
        ; to change the ownership information and money possessed
        [[And  [Retract [Not [Owns ?p ?i]]]
               [Owns ?p ?i]
               [Retract [Wealth ?p ?w*T-Integer]]
               [Wealth ?p ?w'*T-Integer]] <
                     [Not [Owns ?p ?i]]
                     [Buys ?p ?i]
                     [Prove [CanAfford ?p ?i]]
                     [wealth ?p ?w]
                     [Cost ?i ?price*T-Integer]
                     [:= ?w' (- ?w ?price)]
                  :forward [Buys ?p ?i]])
; turn on tracing
RHET -> (Trace-F-Axiom [Buys &rest ?x])
        (([[AND [RETRACT [NOT [OWN ?P ?I]]] [OWN ?P ?I]
               [RETRACT [WEALTH ?P ?W*T-INTEGER]] [WEALTH ?P ?W'*T-INTEGER ]]
               < [NOT [OWN ?P ?I]] [BUYS ?P ?I] [PROVE [CANAFFORD ?P ?I]]
                  [WEALTH ?P ?W*T-INTEGER ] [COST ?I ?PRICE*T-INTEGER ]
                  [:= ?W'*T-INTEGER (- ?W*T-INTEGER  ?PRICE*T-INTEGER )]
               :forward [BUYS ?P ?I]])
; Let's buy A3
RHET -> (Rassert [Buys Jack A3])
                >FC:>Matched and Interpreting Axiom
                   [[AND [RETRACT [NOT [OWN ?P->JACK ?I->A3]]]
                   [OWN ?P->JACK ?I->A3] [RETRACT [WEALTH ?P->JACK ?W*T-INTEGER ]]
                   [WEALTH ?P->JACK ?W'*T-INTEGER ]] < [NOT [OWN ?P->JACK ?I->A3]]
                   [BUYS ?P->JACK ?I->A3] [PROVE [CANAFFORD ?P->JACK ?I->A3]]
                   [WEALTH ?P->JACK ?W*T-INTEGER ] [COST ?I->A3 ?PRICE*T-INTEGER ]
                   [:= ?W'*T-INTEGER  (- ?W*T-INTEGER  ?PRICE*T-INTEGER )]
                   :forward [BUYS ?P->JACK ?I->A3]]
                Matched and Interpreting Axiom [[SBMB [CANAFFORD ?P->JACK ?I->A3]] <
                   [WEALTH ?P->JACK ?W*T-INTEGER ]
                   [COST ?I->A3 ?PRICE*T-INTEGER ]
                   [> ?W*T-INTEGER  ?PRICE*T-INTEGER ]]
```

```
        Axiom Succeeds ([[SBMB [CANAFFORD ?P->JACK ?I->A3]] <
            [WEALTH ?P->JACK ?W->225]
            [COST ?I->A3 ?PRICE->200]
            [> ?W->225 ?PRICE->200]]) : NIL
    ([BUYS JACK A3 <])
; See what Jack owns now
RHET -> (List-Facts [own jack ?i])
    (([OWN JACK A3 <] [OWN JACK A1 <]))
; Now what can be bought? Nothing left!
RHET-> (UnTrace-B-Axiom [CanAfford ?x ?y])
RHET -> (Prove-All [CouldBuy Jack ?i])
    :UNKNOWN
; As noted above, RHET does not maintain truth maintenance information
    ; on builtins so this does not have the desired effect!.
RHET -> (Remove-All [Buys Jack A3])
    NIL
    ; We see that Jack did not get his money back!
RHET -> (Prove [Wealth Jack ?n*T-number])
    [WEALTH JACK ?N->25]
```

3. RHET.3: A Typed Deductive Data Base

RHET.1 introduced a few basic types in RHET. RHET.3 allows users to define their own type system for constants and function terms. The type checking is built-in to the unifier which allows the type system to be used for considerable efficiency gains.

Here are the basic types introduced so far:

:T-U - a RHET object
:T-LISP - a LISP expression
:T-NUMBER - a number, which is subdivided into
    :T-INTEGER - integers
    :T-FLOAT - floating point
    :T-RATIONAL - rational numbers.

These are system defined types. Figure 2 shows the type hierarchy of the most useful pre-defined types. By convention, all RHET types start with the prefix "T-". RHET allows the user to define more complex subtype hierarchies. This includes simple subtype relations as found in many representation systems, and also includes many other relations. RHET does not make any assumptions about type disjointness or force types to be organized into a tree hierarchy. Rather, the user may explicitly declare any set relationship between two types - including disjointness, subtype and overlap - and allows the relationship between two types to be uncertain (e.g. either type :A overlaps type :B or it is a subtype of :B). Figure 3 shows all the possible relationships between two RHET types.

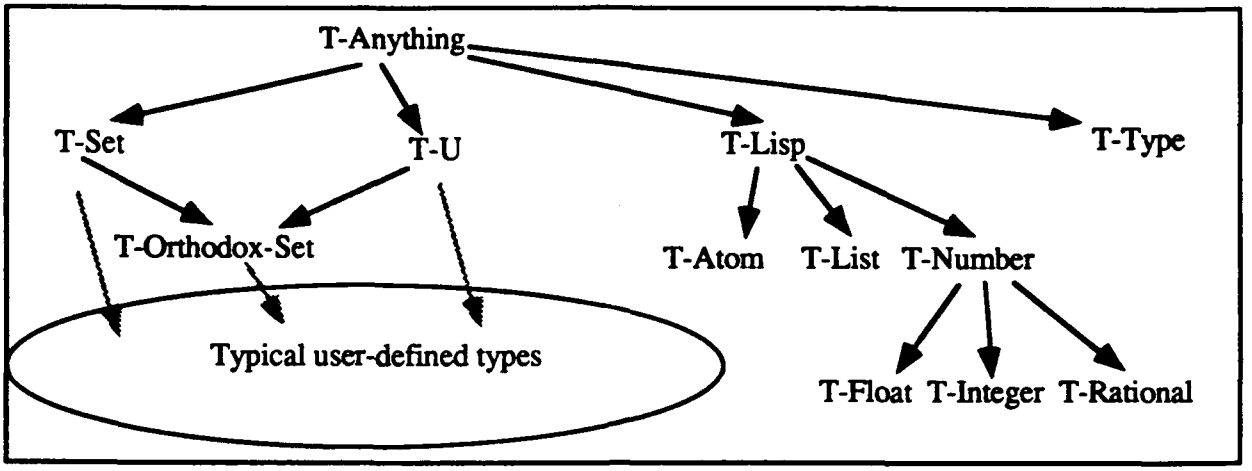


Figure 2: Some pre-defined RHET types

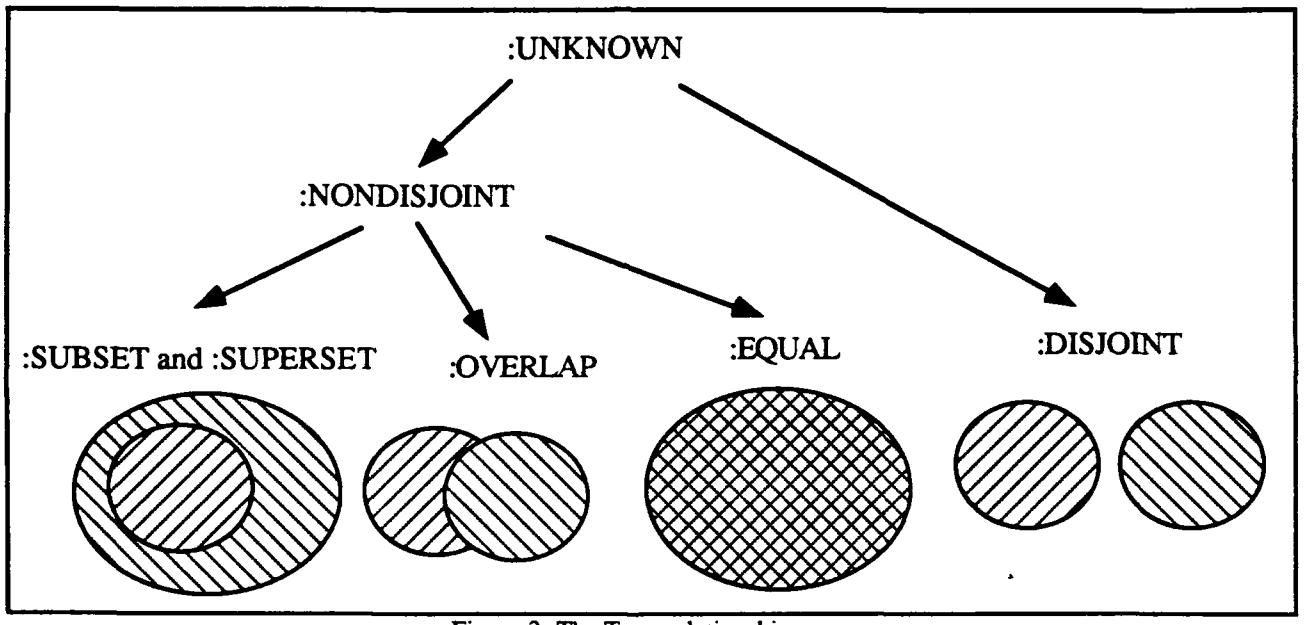


Figure 3: The Type relationships

The system uses constraint propagation techniques to infer the logical consequences of the information the user asserts. For example, if the user asserts that type :T-A is a subset of type :T-B, and that :T-B is a subset of type :T-C, then the system infers that :T-A is a subset of type :T-C. On the other hand, if the user asserts that type :T-D is a subset of type :T-E, and that type :T-F and type :T-D overlap, then the system infers that type :T-D either overlaps or is a subset of type :T-F. Even though this conclusion is disjunctive, it is still useful to the system. In particular, the system knows that, whatever the actual relationship, two variables of type :T-D and type :T-F, respectively, should unify. The resulting variable would have as its type the non-null intersection of type :T-D and type :T-F.

The above example shows the need for RHET to explicitly represent the intersection of types. If :T-A and :T-B are types, then (T-A T-B) is the intersection of the two types. Two variables ?x*T-A and ?y*T-B unify only if the intersection of their types, (T-A T-B), is non-empty. In many cases, the intersection of types can be simplified:

If :T-A is a subset of :T-B, then (T-A T-B) = :T-A

If :T-A is disjoint from :T-B, then (T-A T-B) is the empty set

If :T-A equals :T-B, then (T-A T-B) = :T-A = :T-B.

RHET also supports set subtraction for types. In particular, a term could be defined to be of type :T-ANIMAL but not a :T-DOG. This would be expressed in RHET by the type

(T-ANIMAL - T-DOG).

The general form for a complex type is

(T1 ... Tn - S1 ... Sm)

which is the intersection of T1 through Tn minus the union of S1 through Sm. Using standard set notation this is (T1 $\cap$ T2 .. $\cap$ Tn) - (S1 $\cup$ S2 ... $\cup$ Sm).

Functions can also be typed in RHET, and the type of a specific function term may be dependent on the types of its arguments. For example, the function Spouse might be defined so that if the argument is of type :T-MAN, then the function is :T-WOMAN, whereas if the argument is :T-WOMAN, then the value is :T-MAN. Due to implementation considerations, a function must always first be defined by a most general definition. In the case of Spouse, this might be that an argument of type :T-HUMAN produces a value of type :T-HUMAN. In general, an arbitrary number of type constraints can be defined for a single function, as long as they are consistent with each other. A way to formalize this constraint is to view each type constraint as a horn clause of the form:

Arg1-Constraint ... Argn-Constraint -> Value-Constraint.

For the Spouse function defined above, the constraints would be written as

(C1) :T-HUMAN -> :T-HUMAN
(C2) :T-MAN -> :T-WOMAN
(C3) :T-WOMAN -> :T-MAN

To compute the type of a function [F X] given the definition of F and the type of X, each of the horn clauses defining F whose argument constraint intersects with the type of X contributes a constraint on the value. These constraints on the value are combined by intersection. For example, with the example Spouse, if JACK is a :T-MAN (a subtype of :T-HUMAN), then the type of [Spouse JACK] would be constrained to be of type :T-HUMAN (by rule C1), and of type :T-WOMAN (by rule C2). Thus the type of [Spouse JACK] is (T-WOMAN T-HUMAN) = :T-WOMAN. If we then added that academics only marry academics, i.e.

(C4) :T-ACADEMIC -> :T-ACADEMIC

then if JACK is of type (T-ACADEMIC T-MAN), then the type of [Spouse JACK] would be the intersection of :T-HUMAN (constraint C1), :T-WOMAN (constraint C2) and :T-ACADEMIC (constraint C4), i.e. (T-HUMAN T-WOMAN T-ACADEMIC) = (T-WOMAN T-ACADEMIC).

RHET also can infer the types of arguments given the type of a function. Thus is we unify [Spouse ?x:T-HUMAN] with a variable ?y:T-MAN, then the result will be [Spouse ?x:T-WOMAN]. Similarly, unifying [Spouse ?x:T-HUMAN] with ?y*(T-WOMAN T-ACADEMIC) will produce the desired result, i.e. [Spouse ?x*(T-MAN T-ACADEMIC)].

3.1 Syntax

As already mentioned above, types are atoms and complex types are of the general form (T1 ... Tn - S1 ... Sm). Any type expression can be attached to a variable using the "*" operator creating a variable ranging over that type.

Predicates that take a type descriptor as an argument usually accept either a LISP expression indicating the type, or a variable ranging over the type T-TYPE. For instance, if [Penny] is a constant defined to be in the intersection of :T-PENGUIN and :T-FEMALE, the predicate [Type? [Penny] :T-Penguin] will be provable (since the intersection of :T-PENGUIN and :T-FEMALE is certainly a subtype of :T-PENGUIN). On the other hand, proving [Type? [Penny] ?x*T-TYPE] will bind ?x*T-TYPE to (T-PENGUIN T-FEMALE). A new constant [PennysDouble] could now be defined by proving [Add-Utype ?x [PennysDouble]] (where ?x is bound to (T-PENGUIN T-FEMALE) earlier in the proof). Throughout the definitions below, arguments described as TypeDescriptor will be Lisp atoms or lists (to be converted by RHET into types) or variables of type :T-TYPE. A complex type descriptor is a list of atoms to be converted into types, plus possibly the special atom "-" to indicate type subtraction. Thus the list (A B) is converted into the intersection of :A and :B, while (A B - C) will be converted into the intersection of :A and :B minus the type :C.

The RHET proof strategy assumes a fixed type hierarchy within any proof. Thus, defining a type hierarchy must be done externally using the LISP functions described in the following section

A RHET constant (of type :T-U) may be classified by the user-defined types by asserting type information for it. RHET distinguishes between the immediate type of a constant, which is the most specific type that a constant can have (the **ITYPE**), and a constant simply being of a type in the sense of belonging to the set denoted by the type (a **UTYPE**). A third form of type assertion, called **DTYPE** for **distinguished type**, is used by the equality system as described in RHET.4. As far as RHET.3 is concerned, DTYPE is equivalent to UTYPE. See the RHET.4 tutorial for more information.

The following predicate is convenient for retrieving type information during proofs.

[Type? *Form TypeDescriptor*]

> This predicate is true if the "best" type of the specified form is the type descriptor. The best type is the immediate type if it is defined, or the most specific type that applies to the object if no immediate type is defined. Either argument may be a variable. If the Form is a variable, the predicate will successively return all objects that are defined of the indicated type. If the type descriptor is a variable of type :T-TYPE, it is bound to the best description of the form. For instance, if [Penny] is of type :T-PENGUIN and of type :T-FEMALE, then [Type? [Penny] ?x*T-TYPE] will succeed with ?x bound to (T-PENGUIN T-FEMALE).

Two predicates are provided that allow the user to query the type hierarchy. Since the type hierarchy must remain fixed during any proof, these predicates cannot be asserted.

[SubType? *subType superType*]

This is true if the type denoted by the Lisp expression subtype is a subtype of the type denoted by the lisp expression supertype. Thus, [Subtype? :T-Penguin :T-Bird] would succeed, and [Subtype? ?x*T-Type :T-Bird] could successively return each known subtype of :T-BIRD as it is used in a proof, yielding a new subtype each time it is backtracked to.

[Type-Relation *TypeAtom1 Relation TypeAtom2*]

This predicate succeeds if the relationship between the type denoted by TypeAtom1 and typeAtom2 is the specified relation. The relation returned is an atom as specified in Figure 2, or a type in the case where the intersection is named by the user (see **T-name-intersect**). Thus if :T-PENGUIN is a subtype of :T-BIRD, the [Type-Relation :T-PENGUIN ?x*T-Lisp :T-BIRD] would succeed with the variable ?x bound to :SUBSET. If the type :T-PET either intersects or is a subtype of :T-BIRD, then [Type-Relation :T-PET ?x*T-Lisp :T-BIRD] would succeed with ?x bound to :NONDISJOINT. If the two types are related with a named intersection, then ?x will be bound to the name of the type that is the intersection.

The following predicate that is provided in dealing with types allow the user to create a new constant of a specified type:

[Skolemize *Variable TypeDescriptor*]

Succeeds only if the variable is unbound, and binds the variable to a new constant of the type specified by the type descriptor. It can also be used as a unary predicate: If the type descriptor is not specified, the variable is bound to a new constant consistent with the variable's type.

Note also that the EQ? predicate (see section 4) is useful in many cases for checking types. For instance, if we want to ensure that a variable is of a certain type at some stage of a proof, a simple way to check this is to see if it unifies with a new variable of that type. For example, to ensure that a variable ?x is of type :T-DOG, we could simply prove [EQ? ?x ?newvar*T-DOG].

3.2 Using the Type System

Defining Types

While the type system can be inspected, and the type information for terms can be manipulated by querying and asserting the predicates described above, the type hierarchy itself must be constructed from outside RHET. The following Lisp functions are used to this:

(Tsubtype $typeAtom_0$ $typeAtom_1$ ... $typeAtom_n$)

Asserts that each $typeAtom_i$, for i=1 to n, is a subtype of $typeName_0$.

(Toverlap $typeAtom_1$ ... $typeAtom_n$)

Asserts that each of the specified types pairwise overlap. This is overlap in the :NONDISJOINT sense and does not eliminate the possibility that one type is a subtype of the other.

(Tname-Intersect *NewTypeAtom typeAtom$_1$ ... typeAtom$_n$*)

This is typically used with n=2, and defines the new type NewType to be the named intersection of typeAtom$_1$ and typeAtom$_2$. For example, (Tname-Intersect '2DoorCar '2Door 'Car) would define the type :2DoorCar to be the intersection of :2Door and :Car. This allows the system to simplify complex types of form (2Door Car) to the simple form :2DoorCar. This function can also be used to define the intersection of more than two types as well, although the system must define intermediate types to reduce the n-way intersection to a series of 2-way intersections.

(Tdisjoint *typeAtom$_1$ ... typeAtom$_n$*)

Asserts that all the specified typenames are pairwise disjoint. For example, (Tdisjoint 'A 'B 'C) will assert that :A is disjoint from :B, :B is disjoint from :C and :A is disjoint from :C.

Function types are defined using the following:

(Define-Fn-Type *Fn-Name FunctionSpec**)

This Defines the function named FnAtom to be defined by the set of function specifications, each of form (TypeAtom$_1$ TypeAtom$_2$ ... TypeAtom$_n$) and defines the function with arguments of type Type$_2$ through Type$_n$ to be of type Type$_1$. For example, the function Spouse discussed above would be defined by

(Define-FN-Type 'Spouse '(T-HUMAN T-HUMAN) '(T-MAN T-WOMAN)
'(T-WOMAN T-MAN) '(T-ACADEMIC T-ACADEMIC))

Because of implementation limitations, there are several restrictions of the set of function specifications that are allowed. In particular, the first specification should be the most general declaration (i.e. all subsequent specifications should only involve subtypes of the types used in the first declaration). Second, the rules should be defined so that they are consistent. That is, two intersecting types should not map to types that are disjoint, for then a type in the intersection would not be able to satisfy either constraint. Users interested in defining complex function types should refer to the reference manual.

Defining the Type of Objects

Two lisp functions are provided for defining the types of objects, differing only in whether the type is defined as an immediate type of the object or not.

(Add-IType *TypeAtom* &Rest *Term1 ... Termn*)

This defines the each of the objects listed as terms to have the indicated immediate type. Because of the definition of immediate type, this means that these constants cannot be a member of any subtype of the specified type. Thus if T-BIRD has a subtype T-PENGUIN, (Add-Itype 'T-BIRD [Tweety]) would exclude Tweety from being a Penguin (or any other subtype of T-BIRD). Typically, ITYPE assertions are only used with types that have no subtypes.

(Add-Utype *TypeAtom* &Rest *Term1 ... Termn*)

This defines each of the objects listed as terms to be of the indicated type (i.e. the object is a member of the set denoted by the type). This does not preclude the object being a member of a subtype of the indicated type.

Inspecting Type Information

The following functions are used to delete or inspect the type declarations. The most useful way to inspect the type hierarchy is the graphic display described in the window interface section, if available. A very useful function is one that returns the type of an arbitrary RHET term:

(Type-Object *Term*)

Returns the most specific type for the specified term. If the term contains variables, the most specific type that includes every instantiation of the variables is returned.

(Type-Relation *TypeAtom1 TypeAtom2*)

Returns a keyword or list of keywords indicating the possible relationship(s) between the two specified types.

(Type-Info *TypeAtom*)

Returns a list of relationships between the given type and every other type in the system in the form ((relation1 typeAtom1) ... (relationn typeAtomn))

(Type-Function *FunctionAtom*)

Returns the function specification for the specified function

(Type-Subtype *TypeAtom &key Recursive*)

Returns a list of all immediate subtypes of the specified type. If the recursive option is non-nil, then all subtypes of the type are returned.

(Type-Supertype *typeAtom &key recursive*)

Returns a list of all immediate supertypes, or all supertypes if Recursive is non-nil.

(RTypes)

Returns a list of all types in the system.

Deleting Type Information

The type hierarchy does not support truth maintenance information, so the consequences of adding information about a type cannot be retracted easily. The recommended way to modify the type hierarchy is to completely clear the system using **Reset-Rhet** and re-add the entire new type hierarchy and axioms.

Function definitions are independent of each other so can be incrementally updated. The following functions are provided:

(Clear-All-Fn-Type)

Removes all the function type declarations so far.

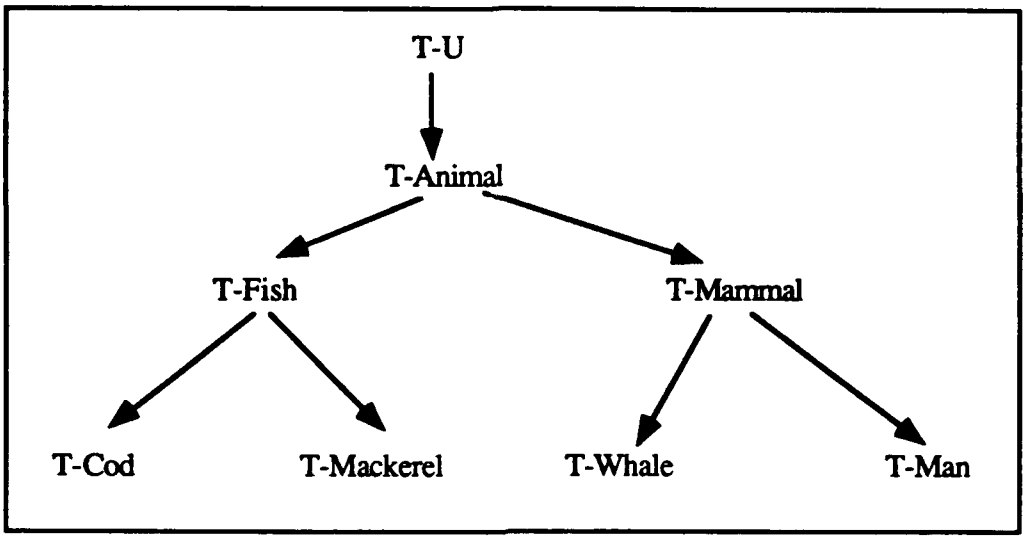


Figure 4: The initial hierarchy

(Remove-Function-Def *FnAtom &Rest FunctionSpec**)

Deletes the specified FunctionSpecs from the definition of the function specified. If no functionSpecs are given, it deletes the entire function definition.

3.3 Examples

These first assertions define a type hierarchy of animals as shown in Figure 4.

```
RHET -> (Tsubtype 'T-U 'T-Animal)
        (NIL)
RHET -> (Tsubtype 'T-animal 'T-fish 'T-mammal)
        (NIL NIL)
RHET -> (Tdisjoint 'T-fish 'T-mammal)
        (T-FISH T-MAMMAL)
RHET -> (Tsubtype 'T-fish 'T-cod 'T-mackerel)
        (NIL NIL)
RHET -> (Tsubtype 'T-mammal 'T-man 'T-whale)
        (NIL NIL)
;System can infer that Mammals are disjoint from Cod:
RHET -> (Type-Relation 'T-mammal 'T-cod)
        :DISJOINT
;Now we add a different hierarchy based on where animals live
RHET -> (tsubtype 'T-Animal 'T-SeaDweller)
        (NIL)
RHET -> (tsubtype 'T-SeaDweller 'T-fish 'T-whale)
        (NIL NIL)
```

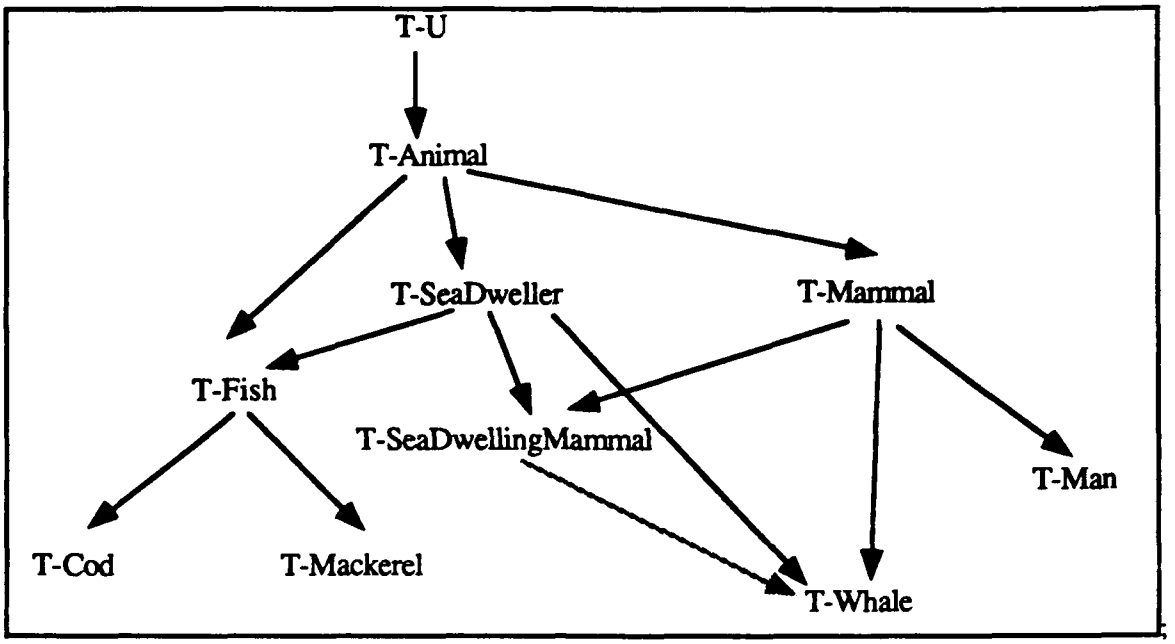


Figure 5: After defining sea dwellers

```
RHET -> (tname-intersect 'T-SeaDwellingMammal 'T-mammal 'T-seadweller)
        NIL
;System can infer that :Cod is a subtype of (T-animal T-SeaDweller):
RHET -> (type-compatiblep 'T-cod '(T-Animal T-seadweller))
        *(T-COD)
; and that whales are sea dwelling mammals
RHET -> (type-compatiblep 'T-whale 'T-SeaDwellingMammal)
        (:SUBSET :EQUAL)
; Let's define some individuals
RHET -> (add-Itype 'T-Cod [Homer])
        ([HOMER])
RHET -> (add-Itype 'T-Whale [Willie])
        ([WILLIE])
;Now some facts about the world
RHET -> (Rassert [[LiveInSea ?x*T-SeaDweller] <1] [[BearLiveYoung ?m*T-mammal] <2])
        (([[SBMB [BEARLIVEYOUNG ?M*T-MAMMAL ]] <2] [[SBMB [LIVEINSEA ?X*T-
              SEADWELLER ]] <1])
RHET -> (Trace-B-Axiom [LiveInSea  &rest ?x] [BearLiveYoung &rest ?x])
        (([[SBMB [BEARLIVEYOUNG ?M*T-MAMMAL ]] <2] [[SBMB [LIVEINSEA ?X*T-
              SEADWELLER ]] <1])
RHET -> (prove [and [liveinsea ?x*T-animal] [bearliveyoung ?x*T-animal]])
        Matched and Interpreting Axiom [[SBMB [LIVEINSEA ?X*T-SEADWELLER ]] <1]
```

```
        Axiom Succeeds ([[SBMB [LIVEINSEA ?X*T-SEADWELLER ]] <1]) : (:TRUE)
        Matched and Interpreting Axiom [[SBMB [BEARLIVEYOUNG ?M*T-SEADWELLINGMAMMAL ]] <2]
        Axiom Succeeds ([[SBMB [BEARLIVEYOUNG ?M*T-SEADWELLINGMAMMAL ]] <2]) : (:TRUE)
        [AND [LIVEINSEA ?X->?M*T-SEADWELLINGMAMMAL ] [BEARLIVEYOUNG ?X-
              >?M*T-SEADWELLINGMAMMAL ]]
;more specifically, willie lives in the sea and bears live young
RHET ->(prove [and [liveinsea [willie]] [bearliveyoung [willie]]])
        Matched and Interpreting Axiom [[SBMB [LIVEINSEA ?X->[WILLIE]]] <1]
        Axiom Succeeds ([[SBMB [LIVEINSEA ?X->[WILLIE]]] <1]) :(:TRUE)
        Matched and Interpreting Axiom [[SBMB [BEARLIVEYOUNG ?M->[WILLIE]]] <2]
        Axiom Succeeds ([[SBMB [BEARLIVEYOUNG ?M->[WILLIE]]] <2]) : (:TRUE)
        [AND [LIVEINSEA [WILLIE]] [BEARLIVEYOUNG [WILLIE]]]
; Now lets define some functions. Men catch SeaDwellers. More specifically
;    whalers catch whales and fishermen catch fish.
RHET ->(tsubtype 'T-man 'T-fisherman 'T-whaler)
        (NIL NIL)
RHET -> (Define-fn-type 'catch '(T-man  T-seadweller) '(T-fisherman  T-fish)
              '(T-whaler T-whale))
        (CATCH (T-MAN T-SEADWELLER) (T-FISHERMAN T-FISH) (T-WHALER
              T-WHALE))
;Might a man catch a whale? Yes, if he's a Whaler.
RHET ->(prove [eq? [catch ?x*T-man] ?y*T-whale])
        [EQ? [CATCH ?X*T-WHALER ] ?Y->[CATCH ?X*T-WHALER ]]
;Define some men
RHET -> (add-Utype 'T-whaler [jack])
        ([JACK])
RHET -> (add-Utype 'T-fisherman [john])
        ([JOHN])
RHET -> (add-Utype 't-man [jake])
        ([JAKE])
;Is whatever Jack catches an animal that bears live young?
RHET -> (prove [bearliveyoung [catch jack]])
        Matched and Interpreting Axiom [[SBMB [BEARLIVEYOUNG ?M->[CATCH JACK]]] <2]
        Axiom Succeeds ([[SBMB [BEARLIVEYOUNG ?M->[CATCH JACK]]] <2]) : (:TRUE)
        [BEARLIVEYOUNG [CATCH JACK]]
;How about John?
RHET -> (prove [bearliveyoung [catch john]])
        :UNKNOWN
```

```
;Does Jake?
RHET -> (prove [bearliveyoung [catch jake]])
        :UNKNOWN
```

Finally, here's the code to test the definition of the Spouse function described above

```
RHET -> (reset-rhet)
        :DEFAULT
RHET -> (Tsubtype 'T-u 'T-human)
        (NIL)
RHET -> (Tsubtype 'T-human 'T-woman 'T-man 'T-academic)
        (NIL NIL NIL)
RHET -> (Tdisjoint 'T-woman 'T-man)
        (T-WOMAN T-MAN)
RHET -> (Toverlap 'T-woman 'T-academic)
        NIL
RHET -> (Toverlap 'T-man 'T-academic)
        NIL
RHET -> (define-fn-type 'spouse '(T-Human T-human) '(T-man T-woman)
        '(T-woman T-man) '(T-academic T-academic))
        (SPOUSE (T-HUMAN T-HUMAN) (T-MAN T-WOMAN) (T-WOMAN T-MAN) (T-
                ACADEMIC T-ACADEMIC))
RHET -> (add-utype 't-man [Jack])
        ([JACK])
RHET -> (prove [Type? [spouse jack] ?x*T-type])
        [TYPE? [SPOUSE JACK] ?X->*T-WOMAN ]
; Here's another way to test the type of [Spouse jack]
RHET -> (prove [eq? [Spouse jack] ?x*T-woman])
        [EQ? [SPOUSE JACK] ?X->[SPOUSE JACK]]
; If we now assert that jack is also an academic, the type of [spouse jack] is refined as well
RHET -> (add-utype 'T-academic [jack])
        ([JACK])
RHET -> (prove [Type? [spouse jack] ?x*T-type])
        [TYPE? [SPOUSE JACK] ?X->*(T-WOMAN T-ACADEMIC ) ]
```

3.4 Additional Functions

(Add-FN-Type *FunctionAtom (Type0 Type1 ... Typen)**)

This allows you to incrementally add to a function definition. See the reference manual for details.

(Type-CompatibleP *TypeDescriptor1 TypeDescriptor2*)

Returns NIL if the first type is not a subset or equal to the second type. Otherwise, it returns the relationship between the types: (:EQUAL), (:SUBSET), (:SUBSET :EQUAL), or a named intersection.

(Type-ExclusiveP *TypeDescriptor1 TypeDescriptor2*)

Returns T if the two types are provably disjoint.

(Type-IntersectP *TypeDescriptor1 TypeDescriptor2*)

Succeeds only if the two types have a named intersection. If the two types intersect, but no name is defined, it returns nil. Thus (Type-IntersectP T1 T2) is not equivalent to (NOT (Type-ExclusiveP T1 T2)).

Finally, two built-in predicates are defined for defining new instances of types as the result of forward chaining. These predicates can only be used for adding information.

[Add-ITYPE *TypeAtom &Rest Term1 ... Termn*]

This predicate defines the listed terms as being of the indicated immediate type. See the lisp function Add-IType for more details. It cannot be used to query the type of an object. To do this, use the builtin predicate **Type?**.

[Add-UTYPE *TypeAtom &Rest Term1 ... Termn*]

This predicate defines the listed terms as being of the indicated type. See the lisp function Add-UType for more details. It cannot be used to query the type of an object. To do this, use the builtin predicate **Type?**.

4. RHET.4: Equality and Inequality

RHET.4 introduces the ability to reason about equality and, conversely, about inequality. Two terms in RHET may be known to be equal, may be known to be distinct (not equal), or not known to be either equal or not-equal. RHET only allows equality assertions between terms of type :T-U, and not between terms of other types such a :T-NUMBER or :T-LISP. RHET.4 uses a generalized unification algorithm: Two terms *t1* and *t2* unify if they can be proven to be equal. If they contain variables, they unify if there is a substitution that makes the two terms equal.

Due to implementation considerations RHET cannot store equality assertions that contain variables. All assertions must involve fully ground terms. Even with this restriction, the situation is complex. For example, if one adds that [F A] is equal to A, then one has actually defined an infinite set of equalities: A equals [F A] equals [F [F A]] equals [F [F [F A]]] and so on. RHET handles such equalities and any other equality assertions possible between two ground terms. In addition, individual equality assertions are not retractable, since there is no truth maintenance facility available for equality assertions (but see contexts in RHET.7).

Terms in RHET can be classified into equivalence classes based on the equality relation. Each equivalence class has one term in its class defined as Primary. This is determined by the system, and typically is the one of the simplest members of the class (i.e. a term that is an atom). RHET's default operation is to use the primary term in the pretty printer for any term in its equivalence class. If this is not desired, set the variable ***Print-FN-Term-Pretty*** to nil.

One may also declare two terms to be not-equal. It would be inconsistent to assert that two

terms are equal if they are already asserted to be not-equal, and vice versa. While individual terms can be explicitly asserted to be non-equal, the more useful way of asserting inequality is by using the type system and declaring them to be members of disjoint types, or by declaring them as distinguished individuals of the same type (see **Add-Dtype** below).

Unification with equality has some significant differences from unification without equality. This is most noticeable when unifying function terms. For example, say we want to unify [P [Mother Jack]] with [P [Mother ?x]]. Under normal unification the most general unifier would have ?x bound to Jack and the result would be [P [Mother Jack]]. But this is not the most general unifier in a system with equality. In particular, if we know that Jack and Jill have the same mother, i.e. [Mother Jack]=[Mother Jill], then Jill is another possible binding for ?x in the unification above! This is a serious problem since the proof strategy depends on being able to find a most general unifier. RHET addresses this problem by allowing the unification but not binding the variable. Rather the variable is constrained so that any binding later assigned must satisfy this unification. In particular, the most general unifier of [P [Mother Jack]] and [P [Mother ?x]] will be

[P [Mother [any ?x1 [EQ [Mother ?x1] [Mother Jack]]]]].

This has the desired effect. The variable ?x1 can be bound to a constant later in the proof only if the equality constraint is satisfied. Such forms, however, can be annoying in final answers, especially when there is only one object in the database that satisfies the constraint anyway. RHET provides a predicate **One-of** that takes such forms containing constrained variables and produces a form with the variable bound to a constant that satisfies the constraint. All of the constants can be obtained eventually by backtracking to this function.

4.1 Syntax

Only minor extensions are required to the RHET syntax as most changes to the system are internal. All the predicates defined previously handle equality automatically. Here we list only a few of the most important ones.

[EQ? *term1 term2*]

Succeeds if the two specified terms are equal, or there is a substitution of variables that makes them equal. The may be used to assert equality relationships between terms of type :T-U. Equality between numbers and other types may be tested but not added. Note the relationship between **EQ?**, **Unify**, and **Identical**:
[Identical t1 t2] entails [Unify t1 t2] entails [EQ? t1 t2].

[Unify *term1 term2*]

Succeeds if the two terms unify without equality, i.e. this is the standard unification algorithm in PROLOG.

[NotEQ? *term1 term2*]

Succeeds if the two terms are provably not equal. This is identical to
[Not [Eq? term1 term2]] but is *not* equivalent to [Unless [EQ? term1 term2]], which is true if RHET cannot prove that term1 equals term2.
Note that the following formulas are all different and are related in the indicated way:
[NotEQ? t1 t2] entails [Unless [EQ? t1 t2]] entails [Unless [Unify t1 t2]] entails [Unless [Identical t1 t2]]

[One-Of *form1 form2*]

Succeeds if *form1* is a ground version of *form2*, *form2* being a function term in the KB. For example: assume we asserted (add-eq [f a] [f b]). Then the result of unifying [f ?x] with [f a] in a proof would bind ?x to [any ?y [eq? [f ?y] [f a]]], e.g.,

RHET -> (prove [eq? [f ?x] [f a]])
[EQ? [F [ANY ?X [EQ? [F ?X] [F B]]]] [F A]].

On the other hand, proving [one-of [f ?z] [f a]]) will first bind ?z to a particular value, e.g.,

RHET -> (prove [one-of [F ?x] [F a]])
[ONE-OF [F ?X->A] [F A]]

RHET -> (prove-all [one-of [F ?x] [F a]])
([ONE-OF [F B] [F A]] [ONE-OF [F A] [F A]]).

A large number of InEQ? relationships can be inferred by RHET as the result of constants being of types that are declared to be disjoint. Objects of the same type can be defined to be not equal using the predicate:

[Add-Dtype TypeDescriptor Term1 ... Termn]

This is equivalent to [Add-Utype Typedescriptor Term1 ... Termn] except that each term is implicitly declared to be non-equal to any other term also declared to have a Dtype of the specified type.

4.2 Using RHET.4

The following functions are available for using the equality subsystem.

Adding Equalities and Inequalities

The direct way to add equalities or inequalities is to simply assert facts involving the **eq?** and **noteq?** predicates. In addition, the following functions are available from the lisp interface. When adding an equality (or inequality) RHET checks for explicit contradictions (i.e. the negation has been asserted) and for type consistency (i.e. two objects with disjoint types cannot be made equal).

(Add-EQ *GroundTerm1 GroundTerm2*)

Adds an equality assertion between the two ground terms if it is consistent to do so. Returns nil if the equality would cause an inconsistency or if the terms are not fully grounded.

(Add-InEq *GroundTerm1 GroundTerm2*)

Adds an inequality assertion between the two ground terms if it consistent to do so. Returns nil if the inequality would cause an inconsistency or if the terms are not fully grounded.

(Add-Dtype *TypeDescriptor Term1 ... Termn*)

This declares the specified terms as distinguished terms of the specified class, the same as the predicate Dtype. Thus each of the terms specified are known to be not equal to any other term declared to be a Dtype of the specified type.

Inspecting the Equality Database

(Equivclass *GroundTerm*)

Returns a list of all ground terms explicitly asserted equal to the specified term. Note that if the variable ***print-fn-term-pretty*** is non-nil, then each member may be printed using the same name. Setting ***print-fn-term-pretty*** to nil prints each term in its original form.

(Equivclass-V *Term*)

This is a version of **Equivclass** that allows variables. It returns a list of all terms that could be equal to the specified term together with the variable binding information required for each term.

(Inequivset *GroundTerm*)

Returns a list of all terms that have been declared explicitly not equal to the specified term. There may be other terms that are provably not equal based on type incompatibility and from **Dtype** assertions.

Tracing Proofs Involving Equality

(Trace-EQ-Object *Term Keyword**)

This function traces changes to equality relationships on the specified term. If no term is specified, all terms are traced. As before, options can be specified that specifying what to trace and what action to take at each trace point. The possible trace points are:

:EQ - Trace changes to the terms equivalence class
:INEQ - Trace changes to the set of terms not equal to traced term
:CLOSURE - Trace changes to any term that references this term

The actions are

:TRACE - print a diagnostic message
:BREAK - invoke a break loop
:DEBUG - invoke the debugger

The default trace settings are :eq :trace :ineq :trace. For example, (Trace-EQ-Object [A]) will cause a message to be printed any time [A]'s equivalence class is changed, or any new inequality assertions are made involving [A].

(Trace-Request *Term**)

This function causes a trace message to be printed every time the equivalence class of one of the indicated terms is involved in an equality assertion. For example:

```
RHET->(trace-request [a])
([A])
RHET->(Rassert [EQ? [f a] [p b]])
Adding Equality between [F A] and [P B] in context SBMB-T
:EQ
RHET->(Rassert [EQ? [p b] [g c]])
Adding Equality between [P B] and [G C] in context SBMB-T
:EQ
```

```
RHET->(Rassert [EQ? [t u] [f g]])
:EQ
RHET->
```

(UnTrace-EQ-Object *Term1 ... Termn*)

Turns off tracing on the specified terms, or on all terms if no argument is specified.

(Untrace-Request *Term**)

Turns off request tracing of the indicated terms.

4.3 Examples

```
; consider two companies and a function that maps employees to their bosses
RHET-> (Tsubtype 'T-U 'T-Human)
        (NIL)
RHET-> (Define-Fn-Type 'Boss '(T-Human T-Human))
        (BOSS (T-HUMAN T-HUMAN))
;  A, B, C, D, E, F, and G are humans. Neither E, F, or G can be
;    equal to each other, and A cannot equal E
RHET-> (Add-IType 'T-Human [A] [B] [C] [D])
        ([A] [B] [C] [D])
RHET-> (Add-Dtype 'T-Human [E] [F] [G])
        ([E] [F] [G])
RHET->(add-Ineq [A] [E])
        T
; Lets add some equalities to define the Boss function
RHET-> (Rassert [EQ? [Boss A] [E]])
        (NIL)
RHET-> (Rassert [EQ? [Boss B] [E]])
        (NIL)
RHET-> (Rassert [EQ? [Boss C] [F]])
        (NIL)
RHET-> (Rassert [EQ? [Boss D] [F]])
        (NIL)
; Some properties
;   E is a generous boss and A, C and G work as programmers
RHET-> (Rassert [[Generous E] <DATA]
                        [[Programmer A] <DATA]
                        [[Programmer C] <DATA]
                        [[Programmer G] <DATA])
```

```
        ([PROGRAMMER G <DATA] [PROGRAMMER C <DATA] [PROGRAMMER A
            <DATA] [GENEROUS E <DATA])
; What is the equivalence class for [Boss A]?
RHET-> (Equivclass [Boss A])
        ([E] [E] [E])
; each is printed using the "best" name - to get the actual terms we set *print-fn-term-pretty*
RHET-> (setq *print-fn-term-pretty* nil)
        NIL
RHET-> (Equivclass [Boss A])
        ([BOSS B] [BOSS A] [E])
; Is anyone who has a generous boss a programmer? Note that
;    the proof uses the any function to construct a most general unifier
RHET -> (trace-request [a] [b] [c] [d] [e] [f] [g])
        ([A] [B] [C] [D] [E] [F] [G])
RHET->(Prove [Generous [Boss ?x*T-Human]] [Programmer ?x])
                Attempt to prove [EQ? [BOSS ?X->A] E]
                Proved [EQ? [BOSS ?X->A] E] as T
                Proved [PROGRAMMER ?X->A] as T
        [AND [GENEROUS [BOSS ?X->A]] [PROGRAMMER ?X->A]]
; Can we add that A equals C? No, because that would entail that
;               [Boss A]=[Boss C] and hence E=F, a contradiction
RHET-> (Rassert [EQ? [A] [C]])
                Adding Equality between [C] and [A] in context SBMB-T
                Debug: When combining [A] and [C] into [C], rhet ran into a problem:
                unioning these classes will force rhet to union [E] and [F] which are not compatible as
                [E] contains [BOSS D] and [F] contains [BOSS B].
                 [condition type: RHET-EQUALITY-PROBLEM]
                Restart actions (select using :continue):
                 0: Back out the entire equality
                 1: Ignore the problem, make things inconsistent, continue adding the equality
                 2: Invoke Rhet debugger
                 3: Retry adding equality of [A] and [C]
        [1] RHET -> :cont 0
        NIL
; So the equality assertion is not allowed even though RHET doesn't explicitly know that [A] is
            ; not equal to [C].
RHET-> (InEquivSet [A])
        ([E])
; Is someone their own boss?  Don't know.
RHET-> (Prove [EQ? [Boss ?x*T-Human] ?x])
```

```
        NIL
; Finally consider the differences between EQ? and unify
; [Boss A] equals [Boss B]:
RHET-> (prove [EQ? [Boss A] [Boss B]])
        [EQ? [BOSS A] [BOSS B]]
; does anyone have the same boss as A:  yes, and an any function is used to produce the most
                ; general unifier
RHET-> (prove [EQ? [Boss A] [Boss ?x*T-Human]])
        [EQ? [BOSS A] [BOSS [ANY ?X*T-HUMAN  [EQ? [BOSS ?X*T-HUMAN] E]]]]
;  who can be the boss of someone -a simple unification allowed by the function typing
RHET-> (prove [EQ? [Boss ?x*T-Human] ?y*T-Human]])
        [EQ? [BOSS ?X*T-HUMAN ] ?Y->[BOSS ?X*T-HUMAN ]]
; Same examples with the unify predicate. Since it does not use equality, the first fails
RHET-> (prove [Unify [Boss A] [Boss B]])
        NIL
;  The second succeeds with ?x bound to A (not shown in answer)
RHET-> (prove [Unify [Boss A] [Boss ?x]])
        T
; and the third succeeds using straight typed unification
RHET-> (prove [Unify [Boss ?x*T-Human] ?y*T-Human])
        [BOSS ?X*T-HUMAN ]
```

5. RHET.5: RHET and LISP

While RHET functions are called from Lisp, we have not yet seen an ability to use Lisp functions during proofs, or to have RHET predicates defined by Lisp expressions rather than axioms. RHET.5 introduces this facility. There are basically three ways to invoke Lisp during a RHET proof. The first involves evaluating a list constructed in RHET as Lisp expression, the second involves defining a RHET predicate by a Lisp function, and the third involves defining a new builtin predicate. This last technique requires a greater understanding of RHET internals and is described in the RHET programmers guide; the others are described here. The techniques differ primarily in the way that backtracking is handled, whether it makes sense to assert such predicates, and how they handle side-effects.

5.1 Predicates that call Lisp

There are three new predicates providing simple ways to evaluate lists as Lisp expressions during a proof. In all cases, any variables in the list are first replaced by their bindings, and then the list is evaluated as a Lisp expression. These lists will be called LispExpressions below even though they technically aren't Lisp expressions since they contain RHET variables.

[Call *LispExpression*]

Rhet replaces any variables in the list with their bindings and then evaluates the list as a Lisp expression. If it returns a non-nil value, the predicate succeeds. If the Lisp expression binds a RHET variable (see example section) during execution, then the RHET variable remains bound as the proof continues. If a proof backtracks to a call, it fails and any variables bound by the function are restored to their original state.

[SetValue *Term LispExpression*]

Succeeds only if the specified term unifies with the value returned by the Lisp expression. If the proof backtracks to this predicate, it fails and the Lisp expression is not re-evaluated. For example, assuming ?y*T-Number is bound to 3, the predicate [SetValue ?x*T-Number (Add1 ?y*T-Number)] would succeed and bind ?x to 4. If ?y was not bound, then this would result in a Lisp execution error since Add1 would have been passed a non-numeric argument. Thus one must either make sure that variables are bound or define Lisp functions that can deal with unbound RHET variables in some way. This will be discussed in more detail below. This predicate can also be used to set multiple variables at once. For details, see the reference manual.

[GenValue *Term LispExpression*]

This predicate is the same as SetValue except that the Lisp expression should return a list of values for the term. The term is unified to the first value on the list and the other values are saved for backtracking, when the term is bound to the next value on the list until the list is exhausted, at which point the predicate fails. Genvalue can be used to recursively invoke the prover. For example,

[Genvalue ?x (Prove-all [P ?y])]

would bind ?x successively to each answer returned by **Prove-all.** This predicate can also be used to set multiple variables at once. For details, see the reference manual.

5.2 Lisp Functions and RHET

The above predicates can be used to evaluate Lisp expressions and bind variables during proofs. They can be tested in axioms just like any other predicate. It does not make sense, however, to add such predicates to the database, so they are not assertable.

A more powerful facility allows the user to define RHET predicates directly by Lisp functions. This also allows the user to define the effects of asserting such a predicate by means of another Lisp function. This capability does not support backtracking, however. To define a predicate in Lisp that supports backtracking, one must define new builtin predicates as described in the programmer's guide.

(Declare-Lispfn *Name Query-Function-Symbol &Optional Assert-Function-Symbol*)

This defines a predicate of the specified name by the specified Lisp functions. Rather than attempting to prove a literal with the specified predicate name, RHET will replace the variables with their bindings and pass the entire variable list to the specified Lisp Query-Function-Symbol as its argument. If all the variables are not bound, then RHET does not call the Lisp function and simply fails. Similarly, if the predicate is asserted, the argument list is passed to the Lisp function Assert-Function-Symbol. If you want to write a Lisp function to manipulate unbound variables, then see the **Call** predicate above.

Of course, all the Lisp functions defined so far can be used in any Lisp function. In addition to those, the following new functions allow the programmer to explicitly manipulate RHET variables and structures in applications of the **Call** predicate.

(Rvariable-P *Lispobject*)

returns non-nil if the argument is an unbound RHET variable

(Type-Object *RhetVar*)

Returns the variable type (see section 4.1).

(Unify *Form1 Form2*)

Unifies the two specified RHET forms, destructively binding any variables (so when the Lisp function returns to RHET the variables have changed). Warning: The programmer must be very careful in binding variables within a Lisp function. It is much better to bind variables using the **SetValue** or **GenValue** builtin predicates if possible. See section 4.1.

(E-Unify *Form1 Form2*)

This is the equivalent of the RHET predicate **EQ?**. Warning: The programmer must be very careful in binding variables within a Lisp function. It is much better to bind variables using the **SetValue** or **GenValue** builtin predicates if possible.

There are also functions that allow a Lisp program to construct new RHET objects. Note that if the reader is set to Rhet mode, RHET objects can be created in Lisp code using the standard reader macros as well: ? constructs a RHET variable, and [...] constructs RHET forms and axioms. But often the programmer will need to create RHET structures dynamically. The following functions allow for this:

(Create-Rvariable *Pretty-Name &Optional (Type *T-U-ITYPE-STRUCT*)*)

Returns a new RHET variable of the indicated name and type, where the type can be constructed using **Make-I-Type**. The pretty-name is the string that will be printed for the variable, and hence should begin with "?".

(Make-I-Type *TypeDescriptor &Optional Permissive*)

Converts the type descriptor into an internal type. Unless Permissive is non-nil, this will return nil unless all types are declared.

(Cons-Rhet-Form *Head &rest Arglist*)

Returns a standard RHET form specified by the arguments. For example,

```
(Cons-RHET-Form 'P 'A)
```

returns

```
[P A],
```

while

```
(Cons-RHET-Form 'P '(A B)
                (Create-Rvariable "?x" (Make-I-Type 'T-Human)))
```

returns

```
[P (A B) ?x*T-Human].
```

(Cons-RHET-Axiom *HeadForm &Rest Form1 ... Formn*)

Returns a standard RHET bc-axiom given a list specification. For example,

```
(Let ((var-x (Create-Rvariable "?x" (Make-I-Type 'T-Human))))
   (Cons-RHET-Axiom (Cons-RHET-Form 'P var-x)
                    (Cons-RHET-Form 'Q var-x)))
```

returns the axiom

```
[[P ?x*T-Human] < [Q ?x*T-Human]]
```

This could now be asserted using **Rassert.**

Here's an example of how one might use these functions to create calls to RHET within a Lisp program and then extract answers. Say we have a Lisp variable *x* and wish to query RHET to find the values of a variable ?y*T-Human such that [P x ?y] is true. We can't simply do

```
(prove [P x ?y])
```

as the "x" will be interpreted as a RHET constant rather than a Lisp variable. Rather we have to build the query using the above functions. But the following will return the value for ?y:

```
(let ((y-var (create-rvariable "?y" (Make-I-Type 'T-Human))))
     (prove (cons-rhet-form 'p x y-var))
     (get-binding y-var))
```

5.3 Examples

Here's a simple example of the use of Setvalue. The predicate P is true of two humans if the first one is greater than twice as rich as the second.

```
RHET -> (Rassert
          ; first define the wealth of various agents
          [Wealth Jake 3]
          [Wealth John 5]
          [Wealth Jason 10]
          ; the definition of P: find the wealth of the first, make sure the variable is bound,
          ; and then multiply it by two using SetValue. Now the comparison can be made.
          [[P ?x ?y] < [Wealth ?x ?w1*T-Number] [Bound ?w1]
              [SetValue ?w*T-Number (* 2 ?w1)]
              [Wealth ?y ?w2*T-Integer] [> ?w2 ?w]])
        ([[SBMB [P ?X ?Y]] < [WEALTH ?X ?W1*T-NUMBER ] [BOUND ?W1*T-NUMBER ]
            [SETVALUE ?W*T-NUMBER  (* 2 ?W1*T-NUMBER )]
            [WEALTH ?Y ?W2*T-INTEGER] [> ?W2*T-INTEGER  ?W*T-NUMBER ]]
        [WEALTH JASON 10 <] [WEALTH JOHN 5 <] [WEALTHJAKE 3 <])
RHET -> (Trace-B-Axiom [P &rest ?x])
        ([[SBMB [P ?X ?Y]] < [WEALTH ?X ?W1*T-NUMBER ] [BOUND ?W1*T-NUMBER ]
            [SETVALUE ?W*T-NUMBER  (* 2 ?W1*T-NUMBER )]
            [WEALTH ?Y ?W2*T-INTEGER ] [> ?W2*T-INTEGER  ?W*T-NUMBER ]])
```

```
RHET -> (Prove [P Jake John])
            Matched and Interpreting Axiom [[SBMB [P ?X->JAKE ?Y->JOHN]] <
                [WEALTH ?X->JAKE ?W1*T-NUMBER ] [BOUND ?W1*T-NUMBER ]
                [SETVALUE ?W*T-NUMBER (* 2 ?W1*T-NUMBER )]
                [WEALTH ?Y->JOHN ?W2*T-INTEGER ] [> ?W2*T-INTEGER ?W*T-NUMBER ]]
            Axiom Fails ([[SBMB [P ?X->JAKE ?Y->JOHN]] < [WEALTH ?X->JAKE ?W1*T-NUMBER ]
                [BOUND ?W1*T-NUMBER ] [SETVALUE ?W*T-NUMBER (* 2 ?W1*T-NUMBER )]
                [WEALTH ?Y->JOHN ?W2*T-INTEGER ] [> ?W2*T-INTEGER ?W*T-NUMBER ]])
    :UNKNOWN
RHET -> (Prove [P Jake Jason])
            Matched and Interpreting Axiom [[SBMB [P ?X->JAKE ?Y->JASON]] <
                [WEALTH ?X->JAKE ?W1*T-NUMBER ] [BOUND ?W1*T-NUMBER ]
                [SETVALUE ?W*T-NUMBER (* 2 ?W1*T-NUMBER )]
                [WEALTH ?Y->JASON ?W2*T-INTEGER ] [> ?W2*T-INTEGER ?W*T-NUMBER ]]
            Axiom Succeeds ([[SBMB [P ?X->JAKE ?Y->JASON]] < [WEALTH ?X->JAKE ?W1->3]
                [BOUND ?W1->3] [SETVALUE ?W->6 (* 2 ?W1->3)] [WEALTH ?Y->JASON ?W2->10]
                [> ?W2->10 ?W->6]]) : NIL
    [P JAKE JASON]
; Consider using Genvalue to find the smallest power of two greater than some given number
        ; (assuming the number is less that 2 to the 8th). We can do this using Genvalue:
RHET -> (Rassert [[SmallestPower ?p*T-Number ?n*T-number] < [GenValue ?p
    (FirstEightPowers)] [> ?p ?n]])
    ([[SBMB [SMALLESTPOWER ?P*T-NUMBER ?N*T-NUMBER ]] <
        [GENVALUE ?P*T-NUMBER (FIRSTEIGHTPOWERS)]
        [> ?P*T-NUMBER ?N*T-NUMBER ]])
RHET -> (Defun FirstEightPowers ()
            (mapcar '(Lambda (x) (Exp 2 x)) '(1 2 3 4 5 6 7 8)))
    FIRSTEIGHTPOWERS
RHET -> (Trace-B-Axiom [SmallestPower &rest ?x])
            ([[SBMB [SMALLESTPOWER ?P*T-NUMBER ?N*T-NUMBER ]] <
                [GENVALUE ?P*T-NUMBER (FIRSTEIGHTPOWERS )]
                [> ?P*T-NUMBER ?N*T-NUMBER ]]
            [[SBMB [P ?X ?Y]] < [WEALTH ?X ?W1*T-NUMBER ]
                [BOUND ?W1*T-NUMBER ]
                [SETVALUE ?W*T-NUMBER (* 2 ?W1*T-NUMBER )]
                [WEALTH ?Y ?W2*T-INTEGER ] [> ?W2*T-INTEGER ?W*T-NUMBER]])
RHET -> (Prove [SmallestPower ?p*T-Number 51])
            Matched and Interpreting Axiom [[SBMB [SMALLESTPOWER ?P*T-NUMBER ?N->51]] <
                [GENVALUE ?P*T-NUMBER (FIRSTEIGHTPOWERS)] [> ?P*T-NUMBER ?N->51]]
            Axiom Succeeds ([[SBMB [SMALLESTPOWER ?P->64 ?N->51]] <
                [GENVALUE ?P->64 (FIRSTEIGHTPOWERS)] [> ?P->64 ?N->51]]) : NIL
    [SMALLESTPOWER ?P->64 51]
```

```
; Here's a variant that returns the power rather than the number, which demonstrates returning
      multiple values:
RHET -> (Rassert [[SmallestExponent ?exp*T-Number ?n*T-number] <
            [GenValue (?v*T-Number ?exp) (FirstEightPowersAndExp)] [> ?v ?n]])
      ([[SBMB [SMALLESTEXPONENT ?EXP*T-NUMBER  ?N*T-NUMBER ]] <
         [GENVALUE (?V*T-NUMBER  ?EXP*T-NUMBER )
            (FIRSTEIGHTPOWERSANDEXP)] [> ?V*T-NUMBER  ?N*T-NUMBER ]])
RHET -> (Defun FirstEightPowersAndExp ()
            (values (mapcar '(Lambda (x) (Exp 2 x)) '(1 2 3 4 5 6 7 8))) '(1 2 3 4 5 6 7 8)))
      FIRSTEIGHTPOWERSANDEXP
; Let's trace a Lisp function
RHET -> (Trace FirstEightPowersAndExp)
      (FIRSTEIGHTPOWERSANDEXP)
RHET -> (Trace-B-Axiom [SmallestExponent &rest ?x)
      ([[SBMB [SMALLESTEXPONENT ?EXP*T-NUMBER  ?N*T-NUMBER ]] <
         [GENVALUE (?V*T-NUMBER  ?EXP*T-NUMBER )
            (FIRSTEIGHTPOWERSANDEXP)]
         [> ?V*T-NUMBER  ?N*T-NUMBER ]]
      [[SBMB [SMALLESTPOWER ?P*T-NUMBER  ?N*T-NUMBER ]] <
         [GENVALUE ?P*T-NUMBER  (FIRSTEIGHTPOWERS)]
         [> ?P*T-NUMBER  ?N*T-NUMBER ]]
      [[SBMB [P ?X ?Y]] < [WEALTH ?X ?W1*T-NUMBER ] [BOUND ?W1*T-NUMBER ]
         [SETVALUE ?W*T-NUMBER  (* 2 ?W1*T-NUMBER )]
          [WEALTH ?Y ?W2*T-INTEGER ] [> ?W2*T-INTEGER  ?W*T-NUMBER ]])
RHET -> (Prove [SmallestExponent ?p*T-Number 51])
            Matched and Interpreting Axiom [[SBMB [SMALLESTEXPONENT ?EXP*T-NUMBER ?N->51]]
               < [GENVALUE (?V*T-NUMBER  ?EXP*T-NUMBER )
               (FIRSTEIGHTPOWERSANDEXP)] [> ?V*T-NUMBER  ?N->51]]
            0: (FIRSTEIGHTPOWERSANDEXP)
            0: returned (2 4 8 16 32 64 128 256) (1 2 3 4 5 6 7 8)
            Axiom Succeeds ([[SBMB [SMALLESTEXPONENT ?EXP->6 ?N->51]] <
               [GENVALUE (?V->64 ?EXP->6) (FIRSTEIGHTPOWERSANDEXP)]
               [> ?V->64 ?N->51]]) : NIL
      [SMALLESTEXPONENT ?P->6 51]
;  Consider an example of a Lisp function that manipulates RHET variables. Say we want to
; define a predicate PlusOne that is true if the first argument is one less than the second argument.
; The predicate fails if both arguments are unbound, but otherwise binds any remaining variables
; appropriately. This would be defined as follows:
```

```
RHET -> (defun plusonefunction (&rest x)
            (cond ((eql (length x) 2)
                   (let ((arg1 (car x))
                         (arg2 (cadr x)))
                     (cond
                   ;If arg1 is an integer variable, and arg2 is a number
                      ((and (rvariable-p arg1)
                            (type-compatiblep 'T-Integer (Type-Object arg1))
                            (numberp arg2))
                       (unify arg1 (- arg2 1)))
                   ;If arg2 is an integer variable, and arg1 is a number
                         ((and (rvariable-p arg2)
                               (type-compatiblep 'T-Integer (Type-Object arg2))
                               (numberp arg1))
                          (Unify arg2 (+ 1 arg1)))
                   ;If both are numbers
                          ((and (numberp arg1) (numberp arg2))
                           (eql (+ 1 arg1) arg2)))))))
      PLUSONEFUNCTION
; Then define a predicate that uses a call to this function
RHET -> (Rassert [[plusone ?x*t-number ?y*t-number] <
                 [call (plusonefunction ?x ?y)]])
      ([[SBMB [PLUSONE ?X*T-NUMBER  ?Y*T-NUMBER ]] <
                 [CALL (PLUSONEFUNCTION ?X*T-NUMBER  ?Y*T-NUMBER )]])
; testing the function - let's trace the Lisp function PlusOneFunction
RHET -> (Trace PlusOnefunction)
      (PLUSONEFUNCTION)
RHET -> (Prove [PlusOne 1 2])
                 0: (PLUSONEFUNCTION 1 2)
                 0: returned T
      [PLUSONE 1 2]
RHET -> (Prove [PlusOne 1 ?x*T-Number])
                 0: (PLUSONEFUNCTION 1 ?Y*T-NUMBER )
                 0: returned 2 ("?Y" . 2)
      [PLUSONE 1 ?X->2]
RHET -> (Untrace)
      NIL
RHET -> (Prove [PlusOne ?x*T-Integer 4])
      [PLUSONE ?X->3 4]
RHET -> (Prove [PlusOne 1 ?x*T-U])        ;fail due to type incompatibility
      :UNKNOWN
RHET -> (Prove [PlusOne ?x*T-Number ?y*T-Number])
      :UNKNOWN
```

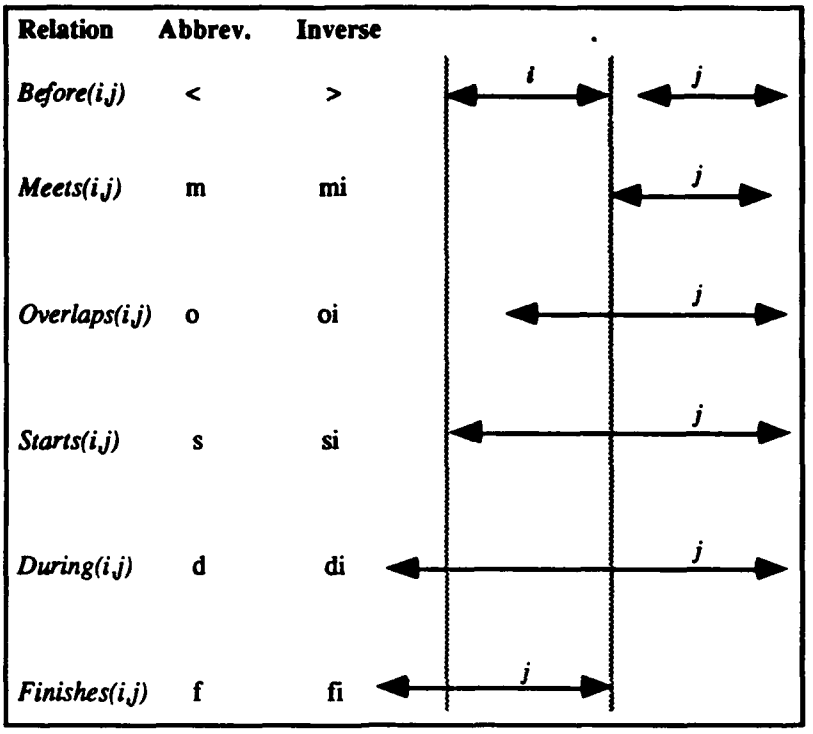


Figure 6: The different interval relationships

6. RHET.6: RHET and Temporal Reasoning

6.1 Basic Concepts

RHET supports temporal reasoning using the TIMELOGIC [Koomen 1988,1989] system as a specialized reasoner. TIMELOGIC implements a version of Allen's interval logic. It allows the user to define constraints between temporal intervals and uses constraint propagation to compute the consequences. As such, TIMELOGIC conceptually integrates with RHET's forward chaining facility. It also supports reference hierarchies, and can automatically generate reference intervals to optimize the network. The interface between TIMELOGIC and RHET supports backtracking over temporal predicates and also supports user contexts as defined in RHET.8.

Allen's Interval Logic takes the temporal interval as primitive and defines 13 possible relations that can hold between intervals. Figure 6 shows six relations and their inverses. Equality is the thirteenth relation. Further details on interval reasoning can be found in Allen [1983]. RHET also provides a limited facility for reasoning about the durations of intervals and about recurrent events. For details, see the reference manual and Koomen [1989].

6.2 Basic Syntax

The principal predicate that forms the interface between RHET and Tempos can be used to add temporal information as well as to retrieve information as needed in proofs. The predicate

[Time-Reln *Time1 Reln Time2*]

:A after	:B before
:C properly contains	:D properly during
:F finishes	:Fi finished by
:M meets	:Mi met by
:O overlaps	:Oi overlapped by
:S starts	:Si started by
:E equals	

Figure 7: The keywords for the temporal relations

is true if and only if the relationship between the two time intervals is the specified relation. The relation may be an atom indicating a single temporal relation, or a list indicating a disjunction of temporal relations. The interval relations are specified the keywords shown in figure 7.

For example, the property [TimeReln T1 :B T2] is true only if time interval T1 is before time interval T2, and [TimeReln T1 (:B :A) T2] is true only if T1 is either before T2 or after T1 (but they do not "overlap" in any way). If [T1] meets [T2], then [Time-Reln T1 (:M :B) T2] will be true since (:M) is a subset of (:M :B).

Often is is useful to check whether a certain relationship between two intervals is possible. Intuitively, we want to check is a certain relationship is one of the remaining possibilities. While we could implement this using the existing RHET functions, this is a common enough operation to warrant its own predicate. The predicate

[Time-Reln-P *Time1 Reln Time2*]

is true if and only if the specified relation is a subset (or equal) to the actual list of possible relationships between the two times. Thus if we had asserted that T1 is before, after or meets T2 (i.e. we added [Time-Reln T1 (:A :B :M) T2]), then [Time-Reln-P T1 :M t2] would be true, as would [Time-Reln-P T1 (:A :M) T2]. For convenience, RHET provides a set of temporal predicates corresponding to the common uses of the **Time-Reln** predicate. These are shown in figure 8.

6.3 Using RHET.6

To use the temporal reasoner, terms must be declared to be of type :T-Time. This may be done using the standard RHET mechanisms, e.g.

(Add-Itype 'T-Time [T1] [T2])

or by using a special function that allows one to define a reference interval for a set of intervals.

(Define-Time *Term1 ... Termn &key Reference-Time-Term*)

This defines the specified terms to be time intervals. If the optional reference interval is specified, then this will be the reference interval for the terms (see [Koomen, 1989] or [Allen, 1983] for more information on reference intervals). For example,

(Define-Time [T1] [T2] [T3])

defines three time intervals in TEMPOS that correspond to three RHET objects automatically declared to be of type T-TIME.

Predicate	Definition
[Time-After T1 T2]	[Time-Reln T1 :A T2]
[Time-Before T1 T2]	[Time-Reln T1 :B T2]
[Time-Contains T1 T2]	[Time-Reln T1 :C T2]
[Time-During T1 T2]	[Time-Reln T1 :D T2]
[Time-Equals T1 T2]	[Time-Reln T1 :E T2]
[Time-Finishes T1 T2]	[Time-Reln T1 :F T2]
[Time-Meets T1 T2]	[Time-Reln T1 :M T2]
[Time-Overlaps T1 T2]	[Time-Reln T1 :O T2]
[Time-Starts T1 T2]	[Time-Reln T1 :S T2]
[Time-Within T1 T2]	[Time-Reln T1 (:D :E :F :S) T2]
[Time-Within! T1 T2]	[Time-Reln T1 (:D :F :S) T2]
[Time-Disjoint T1 T2]	[Time-Reln T1 (:A :B :M :Mi) T2]
[Time-Disjoint! T1 T2]	[Time-Reln T1 (:A :B) T2]
[Time-Intersects T1 T2]	[Time-Reln T1 (:C :D :E :F :Fi :O :Oi :S :Si) T2]
[Time-Starts-Later T1 T2]	[Time-Reln T1 (:A :D :F :Mi :Oi) T2]
[Time-Starts-Earlier T1 T2]	[Time-Reln T1 (:B :C :Fi :M :O) T2]
[Time-Finishes-Later T1 T2]	[Time-Reln T1 (:A :C :Mi :Oi :Si) T2]
[Time-Finishes-Earlier T1 T2]	[Time-Reln T1 (:B :D :M :O :S) T2]

Figure 8: Some temporal predicates defined in terms of Time-Reln

Once intervals are defined, information about them can be added using the **Time-Reln** predicate defined in the previous section.

Time constants can be created dynamically during proofs using the following predicate:

[Time-Skolem ***Variable*** **&optional** ***Reference-Time*****]**

If the variable is bound, then this predicate fails. Otherwise, it creates a new time constant and binds the constant to the variable. If a reference-time is given, it must be already defined as an interval.

Queries and assertions to Tempos can be traced using the following functions.

(Trace-Tempos *&Optional Verbose-P*)

Turns on tracing of all TEMPOS queries and assertions. If *Verbose-P* is non nil, a more detailed trace is given.

(Untrace-Tempos)

Stops tracing Tempos queries and assertions.

Finally, the following function resets the Tempos database:

(Reset-Tempos)

This initializes RHET and the Tempos database.

6.4 Examples

Here are some forward chaining axioms that define the effects of actions in the blocks world.

```
RHET -> (Rassert
            ; If a block ?a is stacked on block ?b at time ?t1, then ?a is on ?b for some time
            ;   ?t2 that immediately follows ?t1.
            [[And [On ?a ?b ?t2] [Time-Reln ?t1 :M ?t2]] <
                [Bound ?t1] [Time-Skolem ?t2] :forward [StackEvent ?a ?b ?t1])
            ; A block cannot be clear and have another block on it at the same time
            [[Time-Reln ?t1 (:A :B) ?t2] <
                [Clear ?b ?t2] [On ?a ?b ?t1] [Bound ?t1] [Bound ?t2]
                :forward [Clear ?b ?t2] [On ?a ?b ?t1]])
        (([[TIME-RELN ?T1 (:A :B) ?T2] < [CLEAR ?B ?T2] [ON ?A ?B ?T1] [BOUND ?T1]
            [BOUND ?T2] :forward [ON ?A ?B ?T1]]
        [[TIME-RELN ?T1 (:A :B) ?T2] < [CLEAR ?B ?T2] [ON ?A ?B ?T1] [BOUND ?T1]
            [BOUND ?T2] :forward [CLEAR ?B ?T2]])
        ([[AND [ON ?A ?B ?T2] [TIME-RELN ?T1 :M ?T2]] < [BOUND ?T1] [TIME-SKOLEM
            ?T2] :forward [STACKEVENT ?A ?B ?T1]]))
; Now let's define some times in a sequence
RHET -> (Define-Time [T1] [T2] [T3])
        ([T1] [T2] [T3])
RHET -> (Rassert [Time-Reln T1 :M T2] [Time-Reln T2 :M T3])
        (NIL NIL)
; If we add that that stack B1 on B2 occurred at time T3
RHET -> (Rassert [StackEvent B1 B2 T3])
        ([STACKEVENT B1 B2 T3 <])
; then we can prove that  B1 is on B2 is true over a time that is met by T3
RHET -> (Prove [And [On B1 B2 ?t] [Time-Reln T3 :M ?t]]))
        [AND  [ON B1 B2 ?T->Time-Skolem-0001]
              [TIME-RELN T3 :M ?T->Time-Skolem-0001]]
;  B1 is on B2 also after T2
RHET -> (Prove [And [On B1 B2 ?t] [Time-Reln ?t :A T2]]))
        [AND  [ON B1 B2 ?T->Time-Skolem-0001]
              [TIME-RELN ?T->Time-Skolem-0001 :A T2]]
; let's add that B2 is clear over a time T5 that is met-by or before T1
RHET -> (define-Time [T5])
        ([T5])
RHET -> (Rassert [Time-Reln T1 (:M :B) T5])
        (NIL)
RHET -> (Rassert [Clear B2 T5])
```

([CLEAR B2 T5 <])

; what is the relationship between T5 and T2?

RHET -> (prove [Time-Reln T5 ?x*T-lisp T2])

[TIME-RELN T5 ?X->(:A :D :E :F :MI :OI :S :SI) T2]

; could T5 be equal to T2 (i.e. could B2 be clear at T2)? yes?

RHET -> (prove [Time-Reln-P T2 :E T5])

[TIME-RELN-P T2 :E T5]

; could T5 be equal to any time where B1 is on B2? no

RHET -> (prove [And [On B1 B2 ?t*T-Time] [Time-Reln-P ?t :E T5])

NIL

RHET.7: RHET as a Frame-based System

7.1 Basic Concepts

Using the tools defined so far, the user could define a frame-like representation system (e.g. [Bobrow and Winograd 1977] [Brachman, Fikes and Levesque, 1983]). This is already done, however, and is provided with the basic system. RHET.7 involves these extensions. The principle concept is that of a structured type. Structured types are like ordinary types except that they carry additional information relevant to each instance of the type. These include such frame-like notions as slots, constraints, and relevant properties.

Slots are simply distinguished function names defined on the type. Thus, saying that the type T-ACTION has an AGENT slot that is filled by a T-HUMAN in a frame system is equivalent to saying that there is a function **f-agent** that maps actions to humans as would be defined by

(Define-Fn-Type 'F-Agent '(T-Human T-Action)).

The notion of filling a slot of an instance A1 of type action with an object JOHN is simply adding the equality [F-Agent A1] = [John].

RHET also distinguishes between types whose instances are fully determined from their role values, the functional types, from those that are not. If T is a **functional type** with role functions r1, ..., rn, then any two instances i1 and i2 of T such that [f-r1 i1]=[f-r1 i2], ..., [f-rn i1]=[f-rn i2] are themselves equal. For example, we might define an action as taking three roles: an agent, a time, and an event (the action consists of the agent causing the event at the specified time). The action type is functional on these three roles - any two actions with the same agent, time and event are the same action.

Types may also have other information associated with them. You can identify properties that must hold of any instance of a given type. These come in two versions: the initializations are RHET facts that are asserted whenever an instance is created. The constraints are RHET forms that must be provably true for any instance. Constraints and initializations may involve a special variable called ?self, which is bound to the new instance to be created. When an instance *i* is created of type *T*, the variable *?self* is bound to *i*, the initializations are added and the constraints are proved. If adding the initializations causes an inconsistency, or a constraint is not true, then the whole operation fails.

RHET also allows the user to define arbitrary classes of relations associated with a type. RHET performs no operations automatically on these relations, but provides tools so the user can define whatever operations they desire. For example, the type :T-action might have relation classes corresponding to the preconditions and effects as used in STRIPS-like planning systems.

Since types may overlap in RHET, an object can be an instance of several structured types at once. In this case, the object must satisfy all the constraints simultaneously. In addition, structured types can be defined as the conjunction of two other structured types. In this case, the new type inherits all the properties of its supertypes. RHET provides a facility to equate role names across different types (i.e. the agent role of type ACTION is the same as the speaker role of type SPEECHACT).

7.2 Basic Syntax

A structured type is implemented by defining two classes of functions that operate on instances of the type. The first are the role functions as described above, and the second, for functional types, is the constructor function.

As an example, consider a type T-Action, that has three roles, an agent of type :T-Human, a time of type :T-Time, and an occurrence of type :T-EVENT. This is represented in RHET using three role functions:

f-agent : from :T-Action to :T-Human
f-time: from :T-Action to :T-Time
f-occurrence: from :T-Action to :T-Event.

If :T-Action is a functional type, it also has a constructor function defined as follows:

c-action: from :T-Human, :T-Event , and :T-Time to :T-Action.

RHET uses an axiom that a constructor function applied to its respective role functions of an instance yields the instance itself, i.e. for any x

[c-action [f-agent x] [f-event x] [f-time x]] = x.

This one axiom yields the required properties for functional types, and RHET adds such an axiom for every instance of the type when it is defined. To see that it works, consider any two instances i1 and i2 with equal role values:

i1 = [c-action [f-agent i1] [f-event i1] [f-time i1]] =
[c-action [f-agent i2] [f-event i2] [f-time i2]] = i2.

Thus given an instance I1 of type T-Action, the appropriate role values can be obtained simply by using the appropriate role function. Say we wanted to assert that the agent role of I1 is [John], and the time role of I1 and I2 are the same, whatever they are. We could add the equalities directly:

(Add-eq [f-agent I1] [John])
(Add-eq [f-time I1] [f-time I2]).

Alternately, role values can be defined using the RHET builtin predicate

[Add-Role Instance *&Rest <Role-Name Value>**]*

which in the above example would be [Add-Role I1 R-Agent John R-time [f-time I2]]. This is

equivalent to simply adding the appropriate equality assertion, but is useful in cases where the role name is not known at the time the code is written and is derived during the proof.

RHET also provides a more general facility for making queries about role values. It provides the predicate

[Role *Instance RoleName Value*]

which is true only if the Instance has the indicated role value. Thus [Role I1 R-agent John] will be true given the above assertions. The rolenames in the predicate are of type T-Atom, and are automatically defined by RHET when the roles are defined for the types. Role names correspond to the function names but begin with an "r-" prefix instead of the 'f-" prefix. This predicate can be used in proofs to find the value of a role (e.g. [Role I1 r-agent ?v]), or finding objects that have certain role values (e.g. [Role ?o r-agent John]). It cannot presently be used to find the rolename relating two objects, although this will be supported in a later version of RHET. More complex queries involving two variables are supported and RHET will iterate through all the possible values as the proof backtracks. This predicate cannot be asserted, but a predicate Add-Role is available as described above.

The type system is assumed to be constant during any given proof, so the facility for defining structured types is primarily available through the Lisp interface as described in the next section.

RHET also provides predicates to allow information about structured types to be queried. The predicate *Subtype?* as described in section 3 allows the users to query subtype relationships, and the following allows for queries about roles:

[Role? *RoleName TypeDescriptor*]

Succeeds only if the specified type descriptor has (explicitly defined or inherited) the specified role. Either argument can be a variable and used to retrieve information about roles that a type has or about types that have a certain role.

7.3 Structured Types and Instances

The principal function for defining structured types is **Define-Subtype** as shown below. It allows the user to define new roles for the type (in addition to the roles the type inherits from its supertype), new constraints, initializations and relations.

(Define-Subtype *Type Supertype &Key (Roles (Rolename Type)*)*)

Define-Subtype defines the specified type as a subtype of the specified supertype. It thus inherits all the roles, constraints, initializations and relations of the supertype. The roles specified may be new roles, or redefinitions of inherited roles to have a more restrictive type. For example,
(Define-Subtype 'T-Event 'T-U :roles '((r-time T-Time)))
would define a type T-Event and a function f-time that defines a time for each event.

(Define-Functional-Subtype *Type Supertype &Key (Roles (Rolename Type)*))*

Define-Functional-Subtype defines a subtype just like Define-subtype except that the new type is functional on its defined roles (see section 7.2). For example, (Define-Functional-Subtype 'T-action 'T-event :roles '((r-agent T-Human) (r-occurrence T-event))) defines a new type T-action that is functional on its roles r-agent, r-occurrence and r-time (inherited from T-event). In other words, for any instance i1 of T-action, i1=[c-action [f-agent i1] [f-occurrence i1] [f-time i1]].

More complex inheritance hierarchies can be created by defining types to be the conjunction of two other types. In this case, the subtype inherits all the properties from both types, including any equality assertions if one or both of the supertypes are functional.

(Define-Conjunction *New-Type (Existing-Type*) &Key :Roles ((New-Role-Name Existing-Role-Name*)*))*

This defines the new type to be a conjunction of the list of specified existing types. It also allows new role names to be defined that are equal to other role names in the existing types. For instance, if :T-Pet is a type with roles R-petname and R-animaltype, and :T-Canine is a type with role R-breed, then we might define pet-dogs as follows

```
(Define-Conjunction 'T-PetDog '(T-Pet T-Canine) :roles
                    '((R-name R-petname) (R-Dogbreed R-breed R-animaltype)).
```

An instance D1 of type :T-PetDog now inherits the properties of both :T-Pet and :T-Canine. Furthermore, we know that [f-name D1]=[f-petname D1] and [f-Dogbreed D1]=[f-breed D1]=[f-animaltype D1].

(Rep-Structures)

This function returns a list of all the structured types that are defined.

Instances of structured types are defined using the functions in this section.The principle function is

(Define-Instance *Instance Type &Rest <Rolename value>*)*

Define-Instance defines an instance of a particular type, optionally specifying role values for the object. For example, (Define-Instance [E1] 'T-event 'R-time [T1]) would define an object [E1] that is an instance of the type T-event and [f-time E1]=T1.

For example, given the definition of T-Action above, we might define a new subtype where the event caused must be a Moving-Event, and involves an object. This could be defined as follows:

```
(Define-Subtype 'T-Moving-Action 'T-Action
        :roles '((R-occurrence T-Moving-Event) (R-object T-Phys-Obj))
```

The entire set of roles for the type T-Moving-Action are now:

(R-agent T-Human) - inherited from T-Action
(R-occurrence T-Moving-Event) - inherited but newly type restricted
(R-Time T-Time) - inherited from T-Action (and hence from T-Event)
(R-Object T-U) - new role

In addition, since T-action is a functional subtype, we know that any instance of T-

Type of Object	Returned multiple values
Constant	:CONSTANT <type> (<role> <value>)*
Rolename	:ROLE-NAME <list of types using this role>
Function name	:FUNCTION-NAME <type restrictions on args> <type of result>
Type	:TYPE-NAME <list of immediate supertypes> <list of roles defined> <list of role type restrictions> <constraints list> <constraints list> <constraints list> <constraints list> <list of initializations> ((<relation-name <relation-form>*)*)
Relation	:RELATION-NAME <list of types using the relation>
Free variable	:VARIABLE <type restriction>
Function w/unbound vars	:FUNCTION <type> (<role> <value>)* <<check???>>
Anything else	:UNKNOWN

Figure 9 The return values of Retrieve-Def

Moving-Action is also an instance of T-Action and thus is functional on its roles r-agent, r-occurrence and r-time. For example, if M1 is an instance of T-Moving-Action, then

[EQ? [c-action [f-agent M1] [f-occurrence M1] [f-time M1]] M1].

A consequence of this is that we could NOT have two Moving-Actions that had identical r-agent, r-event and r-time but different r-objects! If this were possible, then T-Action could not have been a functional type.

Role values for existing objects may be defined using the **Role** predicate, or by adding an equality using the role function to identify the role being defined.

(Retrieve-Def *Object*)

This function returns a description of the specified object, whether it be a type-name, rolename, function name, relation, variable, or constant. The function returns multiple values, the form of which depends on the type of the object and is specified fully in figure 9. For RHET constants, this function returns the constant's type and any role values that are defined. Continuing the example in this section, the call (Retrieve-Def [E1]) would return the values :CONSTANT,T-Event, and (R-Time [T1]). Calling this function with a type name will return all the information about the structured type shown in figure 9. Note that RHET distinguishes four types of constraints and these are listed separately. The typical user can ignore these distinctions. See the reference manual for more details.

If you are introducing objects by means of their constructor functions rather than using **Define-Instance,** then the information about the role values does not become available until RHET *expands* the constructor function. RHET does this automatically when the constructor function is involved in an equality assertion, but the user can force expansion earlier than that using the built-in predicate:

[Expand-Constructor *Constructor-Function*]

This expands the constructor function, i.e., it adds the equalities that define all the role values. For example, given the definition of T-Action above,

[Expand-Constructor [C-Action John E1 T1]]

would create a new constant (say A1) and add the equalities

[f-Agent A1]=[John]
[f-occurrence A1]=[E1]
[f-Time A1]=[T1].

7.4 Constraints, Initializations and Relations

As mentioned above, RHET allows more information to be stored about each type besides the role functions. The *constraints* are properties that must be provably true of an object once it is defined. If a constraint is not true, RHET will add it unless it creates an inconsistency, in which case the instance is not defined. The *initializations* of a type allow for procedural manipulation whenever an object is defined. Finally, types may have arbitrary relations associated with them that are uninterpreted by RHET and can be used for whatever purpose the user wishes. All three of these constructs use a special variable *?self*, which is bound to the instance being defined before any constraint, initialization or relation is created.

The functions **Define-Subtype** and **Define-Functional-Subtype** take optional arguments to specify the initializations, constraints and relations. Thus the full form of **Define-Subtype** is

(Define-Subtype *Type Supertype &Key (Roles (Rolename Type*)) (Relations (Relation-name Relation-Forms*)*) Constraints Initializations*)

Constraints

Constraints are predications that must be true of any instance of the specified type. They can be used to relate role values and for adding any other general knowledge about the instances of the type. In particular, constraints are very useful for capturing equalities between roles, which cannot be stated in general terms since RHET cannot handle equality assertions over non-ground terms. The constraint mechanism allows RHET to add a fully ground equality for each instance as it is defined.

For example, lets us assume that we have a subclass of T-MAN called T-HAPPYMAN, such that each instance of T-HAPPYMAN is known to be happy. In this case, we might have added [Happy ?x*T-HAPPYHUMAN] to the database, but lets see how it also could be realized as a constraint. We would define the type T-HAPPYMAN as follows:

```
(Define-Subtype 'T-HAPPYMAN 'T-MAN
      :constraints '([Happy ?self]))
```

Given this, if we now defined an instance

```
(Define-Instance [H1] 'T*HAPPYMAN)
```

then [Happy H1] would be added to the database. This technique would be preferable over simply asserting [Happy ?x*T-HAPPYMAN] in the following types of situations:

- For technical reasons, say that **Happy** is a builtin, the assertion [Happy ?x]

cannot be directly added to the database. In this case, we get around the problem by adding it for each instance as they are defined.

- If we had forward chaining rules triggered by [Happy ?x], then using the constraint mechanism would trigger forward chaining whenever an instance was defined.

If in adding the constraints an inconsistency was detected then the instance cannot be created and the Define-Instance operation fails. You may use constraints that are not assertable to the database. In this case, RHET tries to prove that the constraint already holds in the database. If it does, then the instance can be created, otherwise the Define-Instance operation will fail.

Initializations

Occasionally, we simply want to run a RHET program whenever an instance of a type is defined. This might be to do some initialization required to create the instance and satisfy the constraints, or to print trace messages, or to do anything else that might be desired. In many cases, one could use constraints or initializations to do the same thing, but there is a significant difference in intent. Constraints are declarative in nature, and indicate things that must be true when the instance has been created. Initializations are generally procedural in nature, and generally are not predicates that will be true after the instance has been created.

For instance, we could print a message whenever an instance is created, or retract some fact, or conditionally add some facts. Here is an example where a message is printed out and we retract the fact that the object being defined is Sad (if it is provably sad before the instance is defined):

```
(Define-Subtype 'T-HAPPYMAN 'T-MAN
        :initializations '([And  [Cond ([Sad ?self] [Retract [Sad ?self]])]
                                 [Rprint "Creating instance of " ?self]] )
        :constraints '([Happy ?self]))
```

Relations

The relations facility provides the user with a capability of storing arbitrary information with the type definitions. For example, a STRIPS-style planner might be implemented in RHET by defining the relations *preconditions, adds* and *deletes* to the T-ACTION type. Relations are Defined to RHET with the following function.

(Define-Rep-Relation *Relation-Keyword &Key Inherit-type)*

This defines the indicated relation keywords so that they are now available to use in defining new subtypes. The inherit type indicates whether the relation is to be inherited to all subtypes of types that have this relation (:inherit), or not (:local). The default value is :inherit. For example,

```
(define-Rep-Relation :precondition)
```

defines a relation keyword :precondition that is inherited to all subtypes of any type that has this relation defined.

Once a relation is defined, it may be used in type definitions. For example, here is a definition of a stacking action (T-STACK) as a subtype of the type T-ACTION.

```
(Define-subtype 'T-STACK  'T-ACTION
                :roles '((R-block1 T-BLOCK) (R-block2 T-BLOCK))
                :relations '((:precondition [Clear [f-block1 ?self]] [Clear [f-block2 ?self]])
                             (:effects [On [f-block1 ?self] [f-block2 ?self]])))
```

To retrieve the relations defined for a particular type, the following function is provided.

(Get-relations *Relation-Keyword Type*)

This returns the list of relations associated with the specified keyword for the specified type. The type may be specified as a lisp symbol, or be an actual type structure built using the **Make-I-Type** function.

Since relations are uninterpreted by RHET, they must be controlled by the user. For techniques for defining ways to manipulate relations, see the reference manual. RHET does provide a few basic predicates for manipulating relations that are useful in most common applications. The following predicates allow the user to access the various relations defined for a type.

[Relation-List *List Relation-Keyword Type*]

This succeeds if the specified list is the relation list of the specified type. This can be used to retrieve the relations of a type if the specified list is a variable. Variables are *not* allowed over the relation-keyword or the type. As an example,

```
[Relation-list ?x*T-list :precondition :T-Stack]
```

would bind ?x to the list ([Clear [f-block1 ?self] [Clear [f-block2 ?self]]) with the definitions above.

[Relation-Form? *Form Relation-Keyword Type*]

This succeeds if the indicated form is an element of the indicated relation list of the specified type. If the form is a variable, this predicate will successively bind it to each form in the list on backtracking. For example,

```
[Relation-Form? ?x :precondition :T-Stack]
```

will succeed and bind ?x to [Clear [f-block1 ?self] on the first call with the definitions above.

A relation retrieved from a type will contain the variable ?self. To bind this variable to a particular instance, the following builtin predicate is defined:

[Bind-Self-to-Instance *Form Instance*]

This binds all ?self terms in ?form to ?instance. You can then do an assert-axiom (or fact) on ?form. For example, the following code would add all of the :effects relations for a particular action [A1] to the database:

```
(Rassert [[Add-Action-Effects ?x*T-Action] <
                [Type? ?x ?type*T-anything]
                [Forall ?form  [Relation-Form? ?form :effects ?type]
                               [Bind-Self-To-Instance ?form ?x]
                               [Assert-Fact ?form]]])
(prove [Add-Action-Effects A1])
```

7.5 Examples

```
; Some initial type hierarchy and instances for the examples
(Tsubtype 'T-U 'T-Animal 'T-Phys-Obj)
        (NIL NIL)
(Tsubtype 'T-Animal 'T-Human)
        (NIL)
(Add-Itype 'T-Human [John] [Jack] [Jill])
        ([JOHN] [JACK] [JILL])
(Add-Itype 'T-Phys-Obj [Box1] [Box2])
        ([BOX1] [BOX2])
(Define-Time [T1] [T2])
        ([T1] [T2])
; Now let's define some structured types
(Define-Subtype 'T-Event 'T-U :roles '((r-time T-Time))
        #<Frame T-EVENT :Roles R-TIME*T-TIME>
(Define-Subtype T-Moving-Event 'T-Event
    :roles '((r-object T-Phys-Obj)))
        #<Frame T-MOVING-EVENT :Roles R-TIME*T-TIME R-OBJECT*T-PHYS-OBJ>
; Let's define an instance of T-Moving-Event
;  with one role specified
(Define-Instance [E1] 'T-Moving-Event 'R-time T1)
        [E1]
(Retrieve-Def [E1])
        :CONSTANT
        T-MOVING-EVENT
        (R-TIME [T1] R-OBJECT [F-OBJECT E1])
; Let's define the second role
(Rassert [EQ? [f-object E1] [Box1]]
        :EQ
(Define-Functional-Subtype 'T-action 'T-event
                    :roles '((r-agent T-Human) (R-occurrence T-event)))
        #<Frame T-ACTION :Roles R-TIME*T-TIME R-AGENT*T-HUMAN
              R-OCCURRENCE*T-EVENT
              :Functional-Super-Type(s) (T-ACTION)>
; Since this is functional, if all three roles were identical, then the instances would be identical
(Define-Instance [A1] 'T-Action 'R-Agent [Jack] 'R-time [T1])
        [A1]
```

```
(Define-Instance [A2] 'T-Action 'R-Agent [John] 'R-time [T1] 'R-occurrence [E1])
        [A2]
(Retrieve-def [A1])
        :CONSTANT
        T-ACTION
        (R-TIME [T1] R-AGENT [JACK] R-OCCURRENCE [F-OCCURRENCE A1])
(prove [Eq? A1 A2])
        NIL
; Now if we add enough information to make the roles equal
RHET ->(add-role [A1] 'R-occurrence [E1])
        [A1]
RHET ->(add-eq  [John] [Jack])
        :EQ
; then the two actions are the same
RHET ->(prove [Eq? A1 A2])
        [EQ? A2 A2]
;  Here's an example defining relations and later instantiating them
RHET -> (Define-rep-relation :effect)
        #<TERM-SUPPORT:REP-RELATIONS @ #x133b71e>
RHET -> (Define-Subtype 'T-NewAct 'T-U :relations '((:effect [P ?self] [Q ?self])))
        #<Frame T-NEWACT
        :Relations (:EFFECT [P ?SELF*T-NEWACT ] [Q ?SELF])>
RHET -> (Define-Instance [A1] 'T-NewAct)
        [A1]
RHET -> (Rassert [[Add-Action-Effects ?x*T-NewAct] <
                        [Type? ?x ?type*T-anything]
                        [Forall ?form       [Relation-Form? ?form :effect ?type]
                                            [Bind-Self-To-Instance ?form ?x]
                                            [Assert-Fact ?form]]])
        ([[SBMB [ADD-ACTION-EFFECTS ?X*T-NEWACT ]] <
                [TYPE? ?X*T-NEWACT  ?TYPE*T-ANYTHING]
                [FORALL! ?FORM
                    [RELATION-FORM? ?FORM :EFFECT ?TYPE*T-ANYTHING ]
                    [BIND-SELF-TO-INSTANCE ?FORM ?X*T-NEWACT ]
                    [ASSERT-FACT ?FORM]]])
RHET -> (list-all [P ?x])
        NIL
RHET -> (prove [Add-Action-Effects A1])
        [ADD-ACTION-EFFECTS A1]
```

RHET -> (list-all [P ?x])
([P A1 <])
RHET -> (list-all [Q ?x])
([Q A1 <])

7.6 Explicit Sets in RHET

RHET provides a limited facility for representing ordered finite sets. An ordered finite set is just a restricted form of a list that can be treated as a first-class object by RHET. In particular, sets can be used in equality assertions whereas lists cannot. As a result, the value of a role function can be a set, but not a list.

Ordered sets are written using curly braces. Thus, {[a] [b] [c]} is an ordered set of three RHET objects, [a], [b] and [c]. A set might also contain variables. For example, {[a] ?x ?y} is a set containing three elements. In addition, a set whose cardinality is not yet known can be indicated as shown in {[a] &rest}, which is a set containing at least one element, namely [a].

Only limited unification is supported for sets. In particular, sets may only unify if they have explicitly the same number of elements, and the elements are in the same order. In fact, thinking of the ordered sets as lists gives the right intuitions. Thus {[a] ?x ?y} will unify with {[a] [b] [c]} with ?x bound to [b] and ?y bound to [c]. {[a] [b]} and {[b] [a]} will *not* unify as the elements are not in the same order. Also, because of implementation considerations, {[a] &rest} will not unify with {[a] [b] [c]} even though intuitively they should unify.

To set up a role function with a set value, we need to be able to define set types. All sets are of type T-SET. Subtypes of T-SET can be defined by restricting the elements of a set to be of some other RHET type. The type T-ORTHODOX-SET is predefined in RHET, and the elements of this set are of type T-U. All user-defined set types whose elements are of type T-U must be subtypes of T-ORTHODOX-SET. For example, we might have a set consisting only of elements of type T-HUMAN. We can define such sets using the following function.

(Define-Set-Type *NewType ParentType ElementType*)

Defines the new name to be a subset of the parent type (which must be a subset or equal to T-SET), where every element in the set is of the specified element type. Thus
(Define-Set-Type 'T-PEOPLE 'T-ORTHODOX-SET 'T-HUMAN)
defines a type T-PEOPLE whose instances are sets of elements of type T-HUMAN.
Given this, then
(Define-Set-Type 'T-BOYSCLUB 'T-PEOPLE 'T-MALE)
defines a type T-BOYSCLUB whose instances are set of elements of type
(T-HUMAN T-MALE).

Given such set types, we can have variables ranging over these sets by using a new syntax. The variable ?x*!T-HUMAN is a variable ranging over sets whose elements are all T-HUMAN.

We also can now define RHET terms as sets using **Add-Itype** and so on. Thus, the result of

(Add-Itype 'T-PEOPLE [P1])

is that [P1] now denotes a set consisting of elements of type T-HUMAN. Of course, given only this definition, RHET has no idea what elements are in this set. The value of the set can be

specified using an equality assertion. Thus to say that [P1] consists of [H1] and [H2], two objects of type T-HUMAN, we simply assert

(Add-Eq [P1] {[H1] [H2]}).

If we know that [H1] and [H2] are in the set, but there might be other elements as well, then we would assert

(Add-Eq [P1] {[H1] [H2] &rest}).

The **Member** predicate is defined on sets as well as lists. If can be used to enumerate the known elements of a set. For example, given the above definitions

(prove-all [Member ?x [P1]])

will successively bind ?x to each element of the set [P1] (provided that the type restrictions on the variable match the types of the elements, of course). The **Member** predicate can also be used to add new elements to a set. Thus if I later find out that [H3] is also in the set, then I can add it by

(Rassert [Member [H3] [P1]]).

RHET allows roles to take multiple values by allowing role function values to be sets (i.e. the function type is a subtype of T-ORTHODOX-SET). For example, we might define a set type consisting of doors, as in

(Define-Set-Type 'T-DOOR-SET 'T-ORTHODOX-SET 'T-DOOR)

We could then define a role function for the type T-CAR that is a set of doors. For example, if [Car1] is a T-CAR, then [f-CarDoor [Car1]] could be a set composed of elements of type T-DOOR. Given this, we might define the R-Door role of [Car1] to consist of two doors, [D1] and [d2] by

(add-eq [f-CarDoor [Car1]] {[D1] [D2]}).

Sets can also be constructed using a variant of **SetValue** which constructs a set out of a list of values returned by a Lisp function.

[Set-SetValue *Variable LispExpression*]

Succeeds only if the variable is bound to the set consisting of the values returned by the Lisp expression. If the proof backtracks to this predicate, it fails and the Lisp expression is not re-evaluated. This predicate also can be used to set multiple variables simultaneously. For details, see the reference manual.

Finally, if I discover that I know all the elements of the set, the I can assert this by fixing the cardinality of the set by asserting:

[Fixed-Cardinality *Set*]

This predicate is true if the specified set has a fixed cardinality (i.e. all its members are known and there are no other members). Asserting this predicate fixes the cardinality of a set to be the current known contents. thus, if a set [P1] is currently equal to {[H1] [H2] [H3] &rest}, then after asserting [Fixed-Cardinality P1], [P1] will be equal to {[H1] [H2] [H3]}.

RHET.8: RHET with Contexts

8.1 Basic Concepts

All assertions in RHET except for type hierarchy declarations can be organized into hierarchical contexts. There are two distinct uses of contexts in RHET. The first allows a user to partition facts and axioms into a tree-structure of databases, where there is an inheritance of facts and axioms from super-contexts as in CONNIVER [Sussman and McDermott 1972]. For example, given the context tree in Figure 10, RHET could prove [S], [R], [Q] and [P] in context C4 (using axioms in T, C2 and C4), whereas RHET could only prove [R], [Q] and [P] in C5, [S], [R] and [P] in C6, and [P] and [R] in C7.

Such contexts are useful for maintaining competing hypotheses for a wide range of reasoning problems. For instance, a plan reasoner might need to maintain several different competing interpretations of an action performed by some other agent. All RHET functions for asserting, deleting or querying facts take an optional argument to indicate the context. In addition, there are procedures to create and delete leaf contexts from the context tree. For example, the left hand branch of the context tree shown in Figure 10 could be constructed from the following assertions (assuming T is the root context).

```
(Rassert [[P] < [Q]] [[R] < [S]] )
(Create-Ucontext 'C2 'T)
(Rassert [[Q] < [R]] :rcontext (ucontext 'C2))
(Create-Ucontext 'C4 'C2)
(Rassert [S] :rcontext (ucontext 'C4))
(Create-Ucontext 'C5 'C2)
(Rassert [R] :rcontext (ucontext 'C5))
```

The function Create-Context will be described fully in the next section.

The other need for contexts is to provide a facility for representing the beliefs of different agents as belief spaces (Cohen 1978). RHET allows separate databases to be defined for each agent, and allows the representation of simple shared beliefs and nested beliefs. While there are clear limitations of this model as a full theory of belief (e.g. see [Konolige 1985], [Haas 1986], [Moore 1985] for richer models), it is sufficient for many practical applications. The belief spaces are organized hierarchically as shown in Figure 11.

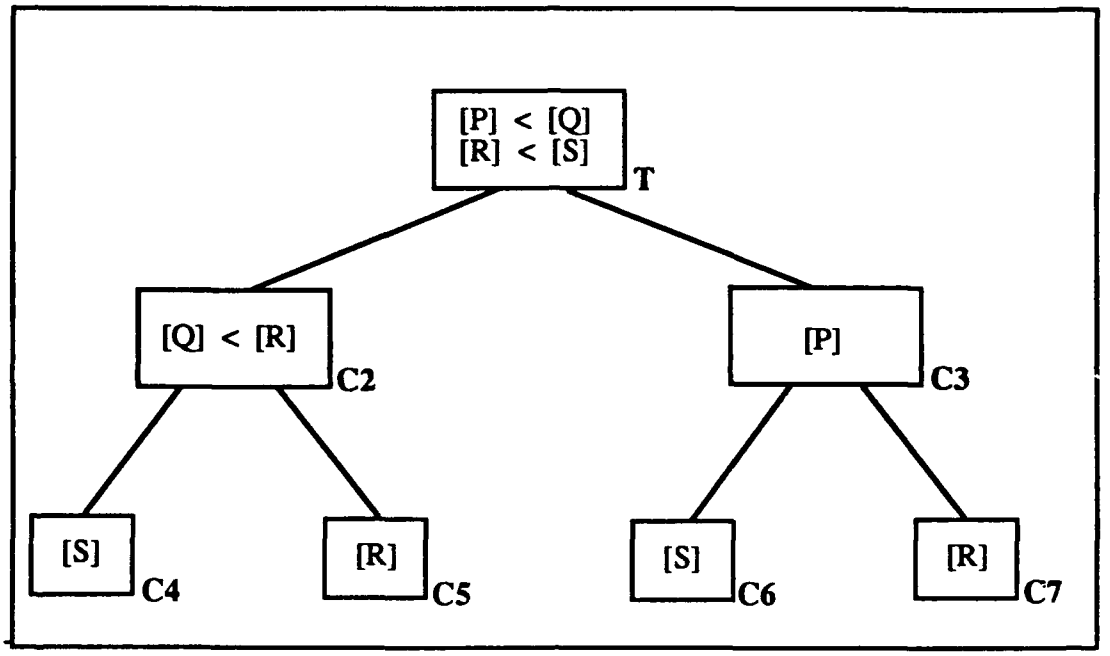


Figure 10: A Context Hierarchy

While the user could set up such a hierarchy explicitly, it is complicated to maintain both a belief hierarchy and a hypothesis hierarchy. RHET provides a facility for combining both and presents a simple user view in terms of a structure called the user context. To the user, a user context acts like a hypothetical context in which assertions and queries about beliefs may be made. RHET provides belief operators that act like modal operators in a logic. In particular, the following is a legal formula:

[<belief modality> <formula>]

where the belief modality is an atom of the form xB, for any letter x. This is a formula that indicates that agent *x* believes the specified fact or axiom. If the letter M is used, then this indicates a mutual belief between all agents. Nested belief modals may be collapsed into single operators, thus [AB [SB [P]]] can be written as [ABSB [P]].

Thus the formula

(1) [AB [[P] < [Q]]]

is interpreted as the system believes agent A believes that [Q] implies [P]. Similarly,

(2) [MB [Q]]

is interpreted as it is shared knowledge among all agents that [Q] holds. Given these two assertions in a user context C1, the formula

(3) [AB [P]]

is provable using the explicit belief (1) and the shared belief (3), whereas [P] (i.e. [MB [P]]) is not provable. While the user works in terms of modal operators, RHET implements the operators in terms of context shifts. Thus asserting (1) is implemented by moving to RHET context AB and adding [P] < [Q]. Likewise, asserting (2) is implemented by moving to RHET context MB and

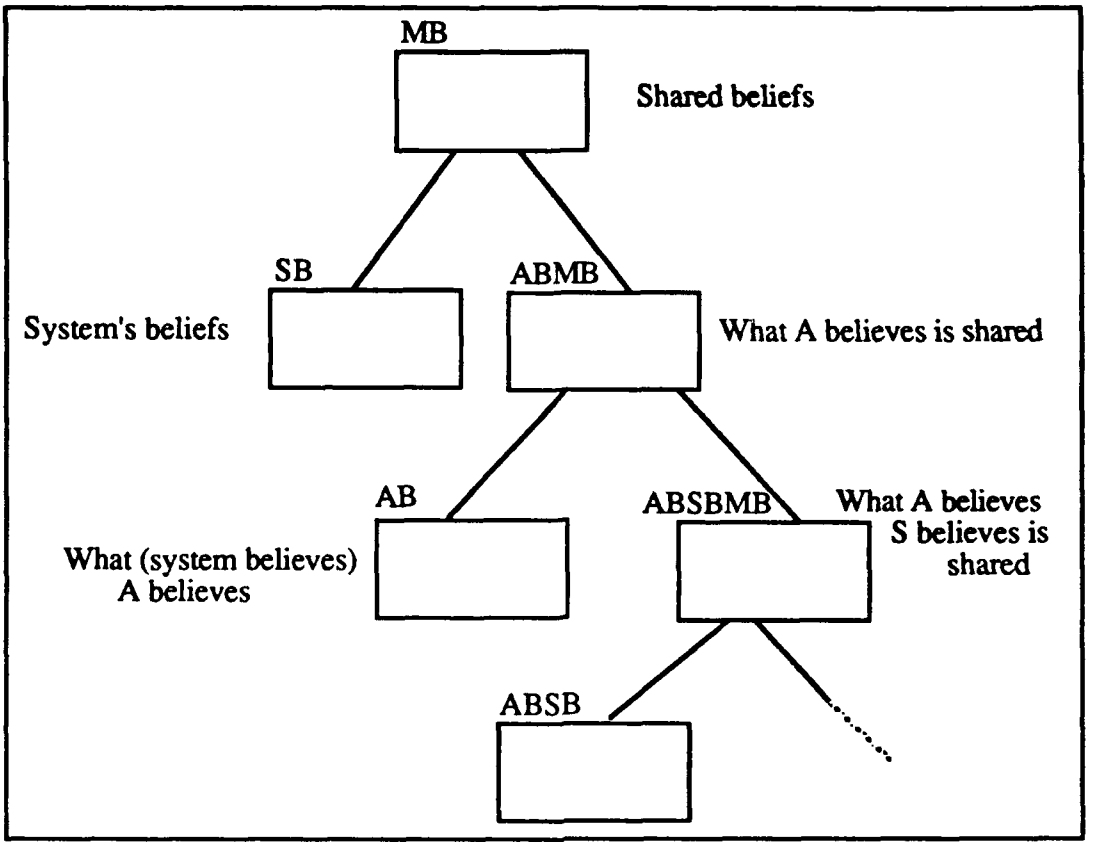


Figure 11: The Belief Space Hierarchy

adding [Q]. Query (3) is implemented by moving to context AB and attempting a proof. Since context AB inherits from context MB (see Figure 11) the proof succeeds.

RHET allows more complex axioms involving individual beliefs. For example, axiom (4)

(4) [[AB[Q]] < [SB[Q]]]

says that A believes [Q] if S believes [Q]. Note that this is not a belief of A, it is a relationship between the beliefs of A and S. It is quite different than (5)

(5) [AB [[P] < [SB[P]]]]

which asserts that A believes [P] if A believes S believes [P]. Axiom (5) does ascribe a belief to A, namely the axiom [[P] < SB[P]].

A user context can be thought of as a set of possible leaves of the belief tree. Within a user context, any combination of individual beliefs may be asserted or queried. For example, assume that axioms (4) and (5) are added to the root user context, and let SB[P] and AB[Q] be added to a subcontext called C1. The user view of this situation and the actual RHET implementation are shown in Figure 12.

As can be seen, to the user it appears that there is a simple two-context hierarchy with modally qualified assertions. In the implementation, however, many things have happened. The context C1 is actually a set of belief spaces inheriting from the leaves of the root belief spaces. In addition, when assertions are added to the appropriate space (based on the belief operators on the left hand side of the axiom), each axiom on the right hand side is indexed by its absolute space,

rather than the relative space in the assertion. In particular, axiom (4) ends up in the space AB as

[P] < [SB! Q].

This is read as follows: [Q] is provable if [Q] can be proved in space SB. The exclamation mark indicates an absolute belief modal (i.e. the correct space is found by starting at the root belief space and shifting wrt S to SB). Axiom 5 in space AB, written at

[P] < [SB P]

shows the use of relative belief scoping. Here the appropriate space is found by shifting (with respect to S) from space AB to space ABSB.

Given this set of assertions, it is possible to prove [AB[P]] in context C1, but not in the root context. Consider the actual RHET operations in performing the proof in C1. The goal is to prove [P] in space ABC1. Axioms (4) and (5) are inherited down to this space and so are applicable. Trying axiom (4), RHET needs to prove [P] in space SBC1. Since [P] is asserted there, the proof succeeds. In the root context, the proof fails because [P] cannot be proven in space SB.

Limitations of the Belief Model

Any RHET formula can be added to a belief space, including equality assertions and type information about individuals. The actual type hierarchy, however, is not expressible in RHET assertions and cannot be made relative to a belief space. In particular, one cannot have the type *T-DOG be a subtype of *T-MAMMAL in one space and not in another. It is possible, however, for an individual, say [FIDO1], to be of type *T-DOG in one space and of type *T-CAT in another.

The major limitations of the belief model arise in the representation of shared beliefs and in the use of variables. Shared beliefs can be asserted only in the root user context. This restriction is essential since the efficiency of the implementation is based on the fact that the contexts are organized as a tree. A space for hypothetical shared knowledge would require a more complex context hierarchy. For the same reason, shared knowledge about an individual's beliefs cannot be represented. Thus, while

[SBMB[P]]

is allowed and asserts that S believes P is shared knowledge among all agents, the formula "[MBSB[P]]" (i.e. it is shared knowledge that S believes P) is *not* allowed in RHET, and yields a syntax error.

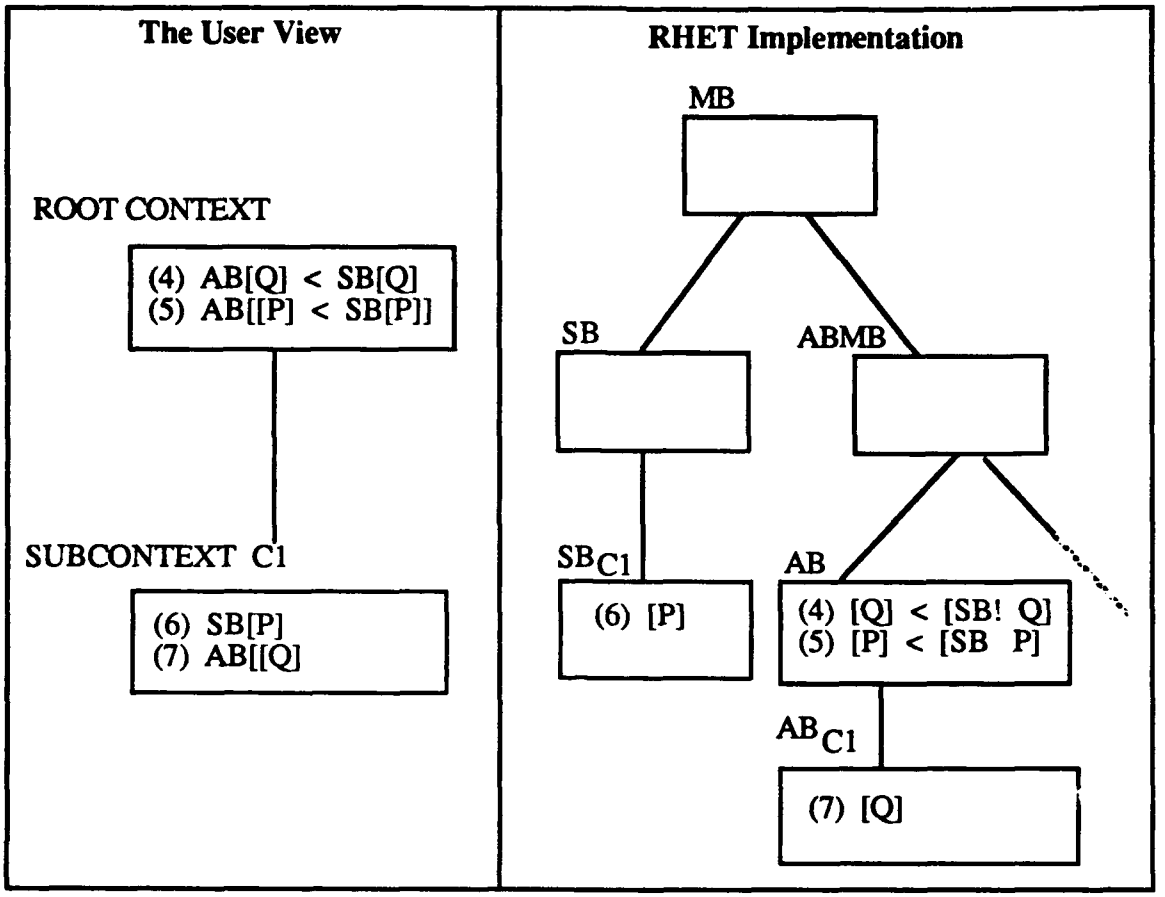


Figure 12: The User View and its RHET Implementation

The other limitation as a belief model concerns variable binding. RHET offers no scoping distinctions for variables across belief operators. For instance, in FOPC and the traditional modal interpretation of belief (e.g. Hintikka, 1969), there is a distinction between

$$\forall x: DOG(x) \supset BEL(A, BARK(x))$$

in which it is asserted that, given any dog, A believes that it barks, and

$$BEL(A, \forall x\ DOG \supset BARK(x))$$

in which it is asserted that A believes that every dog barks. The differences arise when there are actually dogs that A doesn't know about, or when A believes some animal is a dog which in fact isn't. Because there is no mechanism for scoping, RHET can only represent the latter form of formula, in which the type restriction is relative to the belief space the formula is in.

8.2 Using Contexts

As can be seen from the discussion above, there are two factors that determine the actual context that RHET is to perform an operation in: the user context and the belief modality. A single user context and a belief modality form a context called a RHET normal context. These are the contexts that the system actually manipulates. When the user specifies one aspect of a context (say the belief modality), but not the other (the user context), then RHET uses a standard default value for the other to identify the RHET normal context. Unless overridden by the user, RHET uses the default values SBMB for the belief modality, and "T" for the user context. Later on in this section, we will see how these defaults can be changed.

Belief modalities are built into the system and are typically specified using the belief modal construct. User contexts, on the other hand, must be explicitly defined by the user. Here are a set of functions for defining user contexts.

(Create-Ucontext *Name Parent-name*)

This function creates a new user context with the indicated name and with the indicated parent name. Thus (create-Ucontext 'C2 'T) creates a user context named *C2* that is a child of the root context *T*.

(UContext-P *Name*)

Returns the user context indicated by the specified name if one exists, otherwise returns NIL.

(Destroy-Ucontext *Name*)

Destroys the indicated context, including all children of this context.

(Ucontexts)

Returns a list of all defined user contexts.

All functions in RHET that assert, list or remove axioms, or do proofs, take an optional argument to specify the RHET normal context in which the operation is to be performed. If none are specified, RHET uses the default specifications to determine the appropriate RHET normal context. The following functions are used to produce the RHET normal context from user context, a belief modality, or both.

(Get-RNcontext *Name &rest Operator*)

Returns the full RHET normal context with the indicated user context and default belief modality. If the belief modality is not specified, the default modality is used. If the context name is NIL, then the default user context is used. Thus, if the default belief modality is 'SBHB, then (Get-RNcontext 'C2) will return the full RHET normal context SBHB-C2. On the other hand, (Get-RNcontext 'C2 'SB) will return the RHET normal context SB-C2.

In addition, the default context can also be reset.

(Set-Default-Context *RHETcontext)*

Changes the default context to the full RHET context indicated. This can be specified using the **Get-RNcontext** function defined above. For example, to set the default user context to 'C2 and not change the default belief modality, you call

(Set-Default-Context (Get-RNcontext 'C2)).

Now an operation such as

(Rassert [P])

has the exact same effect as

(Rassert [SBHB[P]] :rcontext 'C2).

8.3 More Context Uses

Finally, RHET provides two operators that are convenient for hypothetical reasoning (using user contexts). These can be used to change contexts temporarily during a proof.

[Assume *<Form>**]

This creates a new subcontext to the current user context and then attempts to prove the forms and succeeds or fails just as **And** would. Typically, the first form will actually add information to the new context. The new context is deleted once this proof succeeds or fails. For example, the following formula is true only if [P] implies [Q]:

[Assume [Assert-fact [P]] [Q]].

To prove this, RHET will create a new subcontext, add [P] and then try to prove [Q]. If the original context contained the axiom [[Q] <[P]] then this proof would succeed.

[With *RNcontext <Form>**]

This predicate changes the default context to the specified context and then attempts to prove the indicated forms and succeeds or fails just as **And** would. If the user specifies a name that is not currently a context, the a new context is created with that name.

RHET also provides a function to copy the facts in one context into another context. This is useful when a subcontext has been used for hypothetical reasoning, and now you wish to incorporate the results into the parent context, or when you need to shift facts between belief spaces when modelling communication.

(Copy-Context *Form From-context To-context)*

This function takes all the facts directly (i.e. not inherited) in the *from-context* that unify with the specified form and adds them to the to-context. It does not move axioms. Thus, given

(Rassert [SB [P John]] [MB [P Sue]])

then

(Copy-context [P ?x*T-HUMAN] (Get-RNcontext nil 'SB)
(Get-RNcontext nil 'SBAB))

will add [SBAB [P John]]. Note that [SBAB [P Sue]] is not added here even though [SB [P Sue]] is provable since this fact is inherited from the MB space. All facts in a context may be copied by using a variable as the fact specification as in

(Copy-Context ?x*T-fact (Get-RNcontext nil 'SB)
(Get-RNcontext nil 'SBAB)).

8.4 Examples of Context Use

;the following constructs the database shown in figure 10.

RHET ->(Rassert [[P] < [Q]] [[R] < [S]])

```
        ([[SBMB [R]] < [S]] [[SBMB [P]] < [Q]])
RHET ->(Create-Ucontext 'C2 'T)
        #<UContext C2>
RHET ->(Rassert [[Q] < [R]] :rcontext (Get-RNcontext 'C2))
        ([[SB [Q]] < [R]])
RHET ->(Create-Ucontext 'C4 'C2)
        #<UContext C4>
RHET ->(Rassert [S] :rcontext (Get-RNcontext 'C4))
        ([S <])
RHET ->(create-ucontext 'C5 'c2)
        #<UContext C5>
RHET ->(rassert [r] :rcontext (Get-RNcontext 'c5))
        ([R <])
RHET ->(create-ucontext 'c3 't)
        #<UContext C3>
RHET ->(rassert [p] :rcontext (Get-RNcontext 'c3))
        ([P <])
RHET ->(create-ucontext 'c6 'c3)
        #<UContext C6>
RHET ->(rassert [s] :rcontext (Get-RNcontext 'c6))
        ([S <])
RHET ->(create-ucontext 'c7 'c3)
        #<UContext C7>
RHET ->(rassert [r] :rcontext (Get-RNcontext 'c7))
        ([R <])
RHET ->(prove [p] :rcontext (ucontext 'c7))
        [P]
RHET ->(prove [r] :rcontext (ucontext 'c6))
        [R]
RHET ->(set-default-rcontext (Get-RNcontext 'C7))
        #<context SB-C7>
RHET ->(prove [p])
        [P]
RHET ->(prove [q])
        :UNKNOWN
RHET ->(prove [q] :rcontext (Get-RNcontext 'c5))
        [Q]
;let's add some of the beliefs in figure 13 using the root context T and subcontext
```

```
; C1
RHET -> (reset-rhet)
        :DEFAULT
; then we create an context C1, and (Get-RNcontext 'C1) will tell RHET to update the
; system to include C1 as a context.
RHET ->(create-Ucontext 'C1 't)
        #<UContext C1>
RHET ->(ucontext-P 'c1)
        #<context SB-C1>
;A believes P if A believes that S believes P
RHET ->(Rassert [AB [[P] < [SB P]]]
                ;A believes Q if S believes Q (whether A knows S believes Q or not)
                [[AB [Q]] < [SB! [Q]]])
        ([[SBAB [Q]] < [SB! [Q]]] NIL)
RHET ->(set-default-rcontext  (Get-RNcontext 'C1))
        #<context SB-C1>
;  S believes P and Q in context C1
RHET ->(rassert [sb [p]] [sb [q]])
        (NIL NIL)
; does A believe Q in the root context? no
RHET ->(prove [ab [q]] :rcontext (Get-RNcontext 't))
        NIL
; does A believe Q in context C1 ? yes, since S believes Q in context C1
RHET ->(prove [ab [q]])
        [AB [Q]]
; does A believe P in context C1? no, since A doesn't believe that S believes P
RHET ->(prove [ab [p]])
        NIL
RHET ->(prove [absb [p]])
        NIL
;  lets do that last query another way, specifying the belief modality (and using the default user
        context)
RHET ->(prove [p] :rcontext (Get-RNcontext  nil 'absb))
        :UNKNOWN
; and one other way not using any defaults
RHET ->(prove [p] :rcontext (Get-RNcontext  'c1 'absb))
        :UNKNOWN
;and one final way
RHET ->(set-default-rcontext (Get-RNcontext  'c1 'absb))
```

```
        #<context SBABSB-C1>
RHET ->(prove [p])
        :UNKNOWN
```

Acknowledgements

This work was supported in part by AF-Rome Lab contract number F30602-91-C-0010 and ONR contract N00014-90-J-1811. RHET is the result of nearly a decade of system development, testing and rewriting. There are too many people who have been involved to individually thank them all, so we'll just restrict our thanks to those who directly influenced this document. Many thanks to George Ferguson, Lou Hoebel, Nat Martin and Massimo Poesio, who all carefully read the tutorials and made numerous comments and suggestions for improvements. An thanks to Steve Luk who did the tedious job of individually testing every example in the tutorial, thus finding many errors.

Appendix A Running RHET

Under Allegro

Part of the rhet distribution is a directory marked CL-LIB. A subdirectory, cmu-utils contains defsystem.lisp; a version of the defsystem macro used by rhet and the other knowledge tools in the distribution. Load this, and then either set up a directory for it to look for the defsystem files by default (we use /s5/allegro-local whose files are symbolic links, e.g. rhetorical.system -> /s5/rhet/defsystem.lisp). Alternatively, you can simply load the defsystem file. Then do:

```
(mk:load-system 'rhetorical)
```

and rhet will be loaded.

Once rhet is loaded do:

```
:pa rhet-user
(reset-rhet)
```

and you are ready to begin using rhet. The readtable is globally changed to rhet's readtable by reset-rhet by default under allegro.

If you are going to use the temporal reasoning facilities in RHET, you should load tempos instead. This automatically loads rhet. Thus you do the following:

```
(mk:load-system 'tempos)
:pa rhet-user
(reset-tempos)
```

Warning: if you have tempos loaded, you must use **reset-tempos** rather than **reset-rhet**. Using **reset-rhet** when tempos is loaded will create problems with the type subsystem.

Under Genera

After setting up the proper links in sys:site; just do

```
:load system rhetorical
```

Once the system is loaded you can do

```
:set lisp context rhet
:set package rhet-user
```

in any lisp listener, or

```
<select>-R
```

to bring up a CLIM interface to Rhet. (Unfortunately, since no good lisp listener is provided under CLIM, all the limitations of the one provided are maintained, e.g. the debugger will come up under it's own window, parsing errors typically invoke the debugger, etc.). If you are going to use the temporal reasoning facilities in RHET, you should replace the initial system load above with

```
:load system tempos
```

Appendix B New Function Names in Version 19

With version 19 of RHET, several Lisp functions in the interface were renamed to provide for more consistency in naming. This is the list of new names and the functions that they replaced.

New Name	Old Name
Reset-Rhet	Reset-Rhetorical
Remove-Facts	RetractAll
Remove-Facts-by-index	Clear
Rassert	Assert-Axioms
List-Facts	Find-Facts (but takes one form and returns a simple list)
List-Facts-with-Bindings	Find-Facts-with-Bindings (but takes one form and returns a simple list)
List-Facts-by-index	Find-Facts-by-index
Trace-B-Axiom	Trace-BC-Axiom
Untrace-B-Axiom	Untrace-BC-Axiom
Trace-F-Axiom	Trace-FC-Axiom
Untrace-F-Axiom	Untrace-FC-Axiom.
List-Fact-References	Find-Fact-References
Define-Set-Type	Declare-Set-type
Define-Fn-Type	Declare-Fn-Type
Type-Object	Get-type-object
Type-Relation	Matrix-Relation
Type-Function	Look-up-FN-type
Remove-Function-Def	Delete-fn-type
Remove-Function-Def	Remove-fn-type-def
Get-RNcontext	Ucontext
Get-RNcontext	Operator

References

Allen, J.F. "Maintaining knowledge about temporal intervals", *Comm. ACM 26* (11): 832-843, 1983.

Bobrow, D. and Winograd, T. An overview of KRL: A knowledge representation language, *Cognitive Science 1* (1), 1977.

Brachman, R., Fikes, R. and Levesque, H. KRYPTON: A functional approach to knowledge representation, *IEEE Computer 16* (10), 1983.

Clocksin, W.F. and Mellish, C. *Programming in Prolog*. Berlin: Springer-Verlag, 1981.

Cohen, P.R. On knowing what to say: Planning speech acts, Phd diss. and TR 118, University of Toronto, Computer Science Dept., 1978.

Haas, A. A syntactic theory of belief and action, *Artificial Intelligence 28* (3), 1986.

Hintikka, J. Semantics for propositional attitudes, in J.W. Davis et al (eds), *Philosophical Logic*, Dordrecht: Reidel, 1969.

Konolidge, K. A computational theory of belief introspection, *Proc. IJCAI*, 1985.

Koomen, J.A. The TIMELOGIC Temporal Reasoning System, TR 231 (revised), University of Rochester, Computer Science Dept., 1988.

Koomen, J.A. Reasoning About Recurrence, Phd diss., TR 307, University of Rochester, Computer Science Dept., 1989.

Miller, B. The Rhetorical knowledge representation system reference manual, TR 326, University of Rochester, Computer Science Dept., 1990a.

Miller, B. Rhet programmer's guide, TR 239 (revised), University of Rochester, Computer Science Dept., 1990b.

Moore, R.C. Reasoning about knowledge and action, in J. Hobbs and J. Moore (eds), *Formal Theories of the Commonsense World*, Norwood, N.J.: Ablex, 1985.

Steele, G. *Common Lisp the Language*, 2nd edition, Digital Press, 1990.

Sterling, Leon and Shapiro, Ehud. *The Art of Prolog: Advanced Programming Techniques* . MIT Press Series in Logic Programming, MIT Press, 1986.

Sussman, G. and McDermott, D. "From PLANNER to CONNIVER - a genetic approach", *Proc. of the Fall Joint Computer Conference*, Anaheim, CA, AFIPS Press, 1972.

Index